J

For C Programmers

Henry Rich

2004/10/31

Foreword

You are an experienced C programmer who has heard about J, and you think you'd like to see what it's all about. Congratulations! You have made a decision that will change your programming life, if only you see it through. The purpose of this book is to help you do that.

It won't be easy, and it certainly won't be what you're expecting. You've learned languages before, and you know the drill: find out how variables are declared, learn the syntax for conditionals and loops, learn how to call a function, get a couple of examples to edit, and you're a coder. Fuggeddaboutit! In J, there are no declarations, seldom will you see a loop, and conditionals often go incognito. As for coding from examples, well, most of our examples are only a couple of lines of code—you won't get much momentum from that! You're just going to have to grit your teeth and learn a completely new way to write programs.

Why should you bother? To begin with, for the productivity. J programs are usually a fifth to a tenth as long as corresponding C programs, and along with that economy of expression comes coding speed. Next, for the programming environment: J is an interpreted language, so your programs will never crash, you can modify code while it's running, you don't have to deal with makefiles and linking, and you can test your code simply by entering it at the keyboard and seeing what it does.

If you stick with it, J won't just help the way you code, it'll help the way you think. C is a computer language; it lets you control the things the computer does. J is a language of computation: it lets you describe what needs to be done without getting bogged down in details (but in those details, the efficiency of its algorithms is extraordinary). Because J expressions deal with large blocks of data, you will stop thinking of individual numbers and start thinking at a larger scale. Confronted with a problem, you will immediately break it down into pieces of the proper size and express the solution in J—and if you can express the problem, you have a J program, and your problem is solved.

Unfortunately, it seems to be the case that the more experience you have as a C programmer, the less likely you are to switch to J. This may not be because prolonged exposure to C code limits your vision and contracts the scope of your thinking to the size of a 32-bit word—though studies to check that are still under way and it might be wise for you to stop before it's too late—but because the better you are at C, the more you have to lose by switching to J. You have developed a number of coding habits: for example, how to manage loops to avoid errors at extreme cases; how to manage pointers effectively; how to use type-checking to avoid errors. None of that will be applicable to J. J *will* take advantage of your skill in grasping the essence of a problem—indeed, it will develop that skill considerably by making it easier for you to express what you grasp—but you will go through a period during which it will seem like it takes forever to get things done.

During that period, please remember that to justify your choice of J, you don't have to be as expert in J as you were in C; you only have to be more productive in J than you were in C. That might well happen within a month. After you have fully learned J, it will usually be your first choice for describing a program.

Becoming a J programmer doesn't mean you'll have to give up C completely; every language has its place. In the cases where you want to write code in C (either to use a library you have in C or to write a DLL for a function that is inefficiently computed in J), you will find interfacing J to DLLs to be simple and effective.

This book's goal is to explain rudimentary J using language familiar to a C programmer. After you finish reading it, you should do yourself the honor of carefully reading the J Dictionary, in which you can learn the full language, one of the great creations in computer science and mathematics.

Acknowledgements

I am obliged to the reviewers who commented on earlier versions: Michel Dumontier, Ken Iverson, Fraser Jackson, David Ness, Richard Payne, and Keith Smillie. Brian Schott, Nicholas Spies, and Norman Thomson exchanged emails with me at length to smooth over rough spots. David Steele conducted a painstaking review of several early drafts and suggested many changes great and small. Björn Helgason translated the text into Icelandic, finding a number of errors along the way. Kip Murray's 'review' became more of a dismantling, cleaning, and reassembly operation in which large sections of prose were rewritten as he pointed out to me their essential meaninglessness; the reader should be as grateful to him as I am.

Without the patient explanations of my early teachers in J, Raul Miller and Martin Neitzel, I would have given up on J. I hope that this book pays to others the debt I owe to them.

My current happy career as a J programmer would not have been possible without

the work of the staff at Jsoftware, Inc., who created J. For the patriarch, the late Ken Iverson, I am unworthy to express admiration: I have only awe. I hope his achievement eases the lives of programmers for generations to come. To the rest, both Iversons and non-Iversons, I give my thanks.

The implementation of the J interpreter has required diverse skills: architectural vision, careful selection of algorithms, cold-eyed project management to select features for implementation, robust and efficient coding, performance optimization, and expertise in numerical analysis. Most improbably, all these talents have resided in one man, Roger Hui *il miglior fabbro*. J gives us all a way to have a little of Roger's code in our own. We should aspire no higher.

Change History

2002/6/18: Add chapters on mathematics in J, and section on Symbols; minor changes to wording; bring text up to J Release 5.01

2002/8/16: Minor additions; added section on aliasing; added chapter on sockets

2002/9/26: Added sections on fndisplay, integrated rank support, and ordering of implied loops

2002/11/14 Added explanatory sections, section on the J Performance Monitor, and chapter on Error Messages

2003/07/18 (J5.03) added section on I., updated f. Added chapter on frequent errors. Added section on run-length decoding

2004/10/31 (J5.04) ^:a:, vector cut. Added chapter on Sequential Machines

---

Contents

---

1. Introduction

This book will tell you enough about J for you to use it as a language for developing serious applications, but it is about more than learning the J language: it is also about 'thinking big' in programming, and how programming in J is fundamentally different from programming in C. C programs deal intimately with scalars (single numbers and characters), and even when they combine those scalars into arrays and structures, the operations on the arrays and structures are defined by operations on the scalars. To ensure that each item of an array is operated on, loops are created that visit each element of the array and perform a scalar operation on the element.

Writing code in a scalar language makes you rather like a general who gives orders to his troops by visiting each one and whispering in his ear. That touch-of-Harry kind of generalling can achieve victorious results, and it has the advantage that the orders can be tailored to the man, but its disadvantages are significant: the general spends much mental energy in formulating individual orders and much breath in communicating them individually; more significant, his limited attention is drawn toward individuals and away from the army as a whole. Even the great Rommel was overtaxed at times.

The J programmer is, in contrast, a general who stands before his army and snaps out orders to the army as a whole. Every man receives the same order, but the order itself contains enough detail for the individual men to act appropriately. Such a general can command a corps as easily as a platoon, and always has the 'big picture' in mind.

OK, maybe you're not Rommel, but you are a working programmer, and you suspect that very few practical programs can be represented as array operations—matrix multiplication maybe, or adding a list of numbers—and that, even if a wide range of programs were possible, the set of operations supported must be too vast to be practical: wouldn't we need an array operation for every possible program?

The first half of this book is devoted to showing you that it is indeed possible to write meaningful programs with array operations. We take the approach of looking at the different ways loops are used, and seeing what facilities J has for producing the same result using array operations. We will find that J contains a couple of dozen array-processing primitives and a dozen or so very cleverly chosen pipe-fittings that allow those primitives to be connected together to provide the limitless supply of array-processing functions needed for practical programming.

Interspersed with the elaboration of more and more intricate array operations are treatments of other matters of use in practical programming: structure definition, input and output, performance measurement, calling DLLs, modular programming. Eventually we will see how to use if-then-else and do-while in J, though you will have learned more elegant ways to get the same results.

The last portion of the book is devoted to the optional topic of *tacit programming*, J's language for functional programming. Tacit programming is an extremely terse way of expressing algorithms, one that allows programs that have been compressed from a page of C into a few lines of J to be compressed still further.

<< >> Contents Help

Programming In J

<< >> Contents Help

2. Preliminaries

Notation

C code is set in Arial font, like this: **for(I = 0;I<10;I++)p[I] = q;**

J code is set in Courier New font, like this: **`p =. 10 $ q`**

When J and C use different words for the same idea, the J word is used. The first few times, the C word may be given in parentheses, in Arial font: verb (**function**). When a word is given a formal definition, it is set in bold italics: ***verb***.

Terminology

To describe the elements of programming, J uses a vocabulary that will be familiar, though possibly frightening: the vocabulary of English grammar. We will speak of nouns, verbs, and the like. Don't worry, you're not going to have to write a book report!

Use of this terminology is not as strange as it may seem. Take 'verb', for example, an idea that corresponds to the C '**function**' or '**operator**'. Why not just say 'operator'? Well, that word is also used in mathematics and physics, with a meaning quite different from C's. Even a C 'function' is not a true mathematical function—it can return different values after invocations with the same arguments.

J avoids imprecise usage by choosing a familiar set of words and giving them entirely new meanings. Since J is a language, the vocabulary chosen is that of English grammar. It is hoped that the familiarity of the words will provide some mnemonic value, but as long as you learn the J meanings you are free to forget the grammatical ones. The following table may help:

J word	C word
verb	**function** or **operator**
noun	**object**
copula	**assignment**
punctuation	**separator**

adverb	(untranslatable)
conjunction	(untranslatable)

In keeping with the grammatical flavor of the vocabulary, we say that every ***word*** (**token**) in a J program has a ***part of speech*** (**name type**) which is one of the following: noun, verb, adverb, adjective, copula, or punctuation.

The ***primary parts of speech*** are noun, verb, adverb, and conjunction. Every name we can create, and every word defined by J except for the copulas (**`=.`** and **`=:`**) and punctuation, will be a definite one of the primary parts of speech. In this book, the term ***entity*** is used to mean something that can be any of the primary parts of speech. An entity can be assigned to a name, but most entities are ***anonymous***, appearing and disappearing during the execution of a single sentence (just like intermediate results in the evaluation of C expressions).

A ***noun*** holds data; a ***verb*** operates on one or two nouns to produce a result which is a noun; an ***adverb*** operates on one noun or verb to produce a ***derived entity***; a ***conjunction*** operates on two nouns or verbs to produce a derived entity. Adverbs and conjunctions are called ***modifiers***.

A word on ***punctuation*** under J's definition: it consists of the characters **`( ) '`** and end-of-line (written **`LF`** but representing either a single LF character or the CRLF combination), along with the comment delimiter **`NB.`** and a few other special words like **`if.`** and **`case.`** . There are a lot of other characters that you think of as punctuation, namely **`[ ] , . " ; { }`**, that J uses to do work. You will be especially surprised to find that **`[ ]`** and **`{ }`** are independent rather than matched pairs, but you'll get used to it.

Sentences (statements)

The executable unit in J is called the ***sentence***, corresponding to the C **statement**. The ***sentence delimiters*** in J (corresponding to the semicolon in C) are the linefeed **`LF`** and the control words like **`if.`** that we will learn about later. A sentence comprises all the characters between sentence delimiters; since **`LF`** is a sentence delimiter, it follows that a J sentence must all fit on one line. There is nothing corresponding to **\<CR>** in C that allows you to split a sentence across lines.

All comments start with **`NB.`** and run to the next **`LF`** . The comment is ignored when the sentence is executed.

Word Formation (tokenizing rules)

J's ***name***s (**identifiers**) are formed much as in C. Names must begin with an alphabetic, underscore is allowed, and upper- and lowercase letters are distinguished. Names that end with an underscore or contain two consecutive underscores are special, and you should avoid them until you know what a locale is.

The ASCII graphic characters ('+', for example) are called ***primitives*** (**operators**) in J. You will learn their meanings as we go on.

Any name or primitive (**identifier** or **operator**) can be made into a new primitive by adding **`'.'`** or **`':'`** at the end. Since all primitives are system-defined (i. e. they are **reserved words**), you may not put **'.'** or **':'** in your names. No space is required after a primitive. The part of speech for each primitive is fixed. Example primitives are:

```
+ +. +: { {: {:: i. i: for. select. case. end.
```

The first step in processing a sentence is to split it into ***words***. The words correspond roughly to C **tokens**, after making allowance for the special status of the **`'.'`** and **`':'`** characters. The space and **`TAB`** characters are treated as **whitespace**. One big surprise will be that a sequence of numbers separated by spaces is treated as a single word which denotes the entire list of numbers.

We will be careful to distinguish periods used for English punctuation from the dot that may be at the end of a primitive. When a J word comes at the end of an English sentence, we will be sure to leave a space before the period. For example, the verb for Boolean Or is **`+.`**, while the verb for addition is **`+`** .

Numbers

You do not need to trouble yourself with the distinction between integers, floats, and complex numbers. If it's a number, J will handle it properly. There are a great many ways to specify numbers; consult the Dictionary to learn details, including, among other things, complex numbers, extended-precision integers, and exponential forms. Example numbers are:

`2`

`_2` (underscore, not -, is the negative sign)

`0.5` (since '.' is special, it must not be the first character of a number)

`1e2`

`16b1f` (equivalent to **0x1f**)

_ (infinity)

__ (negative infinity, represented by two underscores)

A noun whose value is one of the numbers 0 and 1 is said to be ***Boolean***. Many verbs in J are designed to use or produce Boolean values, with 0 meaning false and 1 meaning true, but there is no Boolean data type: any noun can be used as a Boolean if its values are 0 or 1.

A word is in order in defense of the underscore as the negative sign. `-x` means 'take the negative of the number `x`'; likewise `-5` means 'take the negative of the number `5`'. In J, the number 'negative 5' is no cloistered companion, accessible only by reference to the number 5: it is a number in its own right and it deserves its own symbol: `_5`.

Characters

An ASCII string enclosed in single quotes is a constant of character type (examples: `'a'`, `'abc'`). There is no notation to make the distinction between C's single-quoted character constants and double-quoted character strings.

There are no special escape sequences such as **'\n'**. If you need a quote character inside a string, double the quote: `'cannot can be shortened to can''t'`. Character constants do not include a trailing NUL (**\0**) character, and NUL is a legal character within a string.

Valence of Verbs (Binary and Unary Operators)

C operators can be unary or binary depending on whether they have one or two operands; for example, the unary ***** operator means pointer dereferencing (***p**), while the binary ***** operator means multiplication (**x*y**).

Similarly, when a J verb (**function** or **operator**) is executed with only one operand (i. e. without a noun or phrase that evaluates to a noun on its left) we say its invocation is ***monadic*** (**unary**); if there is a noun or noun-phrase on its left, that noun becomes a second operand to the verb and we say that the invocation is ***dyadic*** (**binary**). In the case of programmer-defined verbs (**functions**), the versions handling the two cases are defined independently. We use the term ***valence*** to describe the number of operands expected by a verb-definition: a verb-definition has ***monadic valence*** if it can be applied only monadically, ***dyadic valence*** if it can be applied only dyadically, and ***dual valence*** if it can be applied either way. Since the definitions of the monadic and dyadic forms of a verb can be wildly different, when we name a verb we will be careful to indicate which version we are talking about: 'monad **`$`**', 'dyad **`i.`**'.

Note that it is impossible to invoke a verb with no operands. In C we can write **func()**, but in J we always must give an operand.

Note also that the syntax of J limits verbs (**functions**) to at most two operands. When you need a verb with more than two operands, you will represent it as a monad or dyad in which one of the verb's syntactic operands is an aggregate of the actual operands the verb will use during its execution. The first thing the verb will do is to split its operand into the individual pieces. J has primitives to make this process easy.

The value produced by any entity when it is applied to its operand(s) is called its ***result*** (**returned value**).

How Names (Identifiers) Get Assigned

Assignment in J is performed by expressions of the form

```
name =. entity  NB. private
```

and

```
name =: entity  NB. public
```

Names assigned by public assignment are visible outside the entity in which they are defined; names assigned by private assignment usually are not; we will learn the details when we discuss modular code. The difference between the two forms of assignment is in the character following the **`=`** . Just as in C, the assignment expression is considered to produce as its result the value that was assigned, so expressions like

```
a =. 1 + b =. 5
```

are legal. J calls **`=.`** and **`=:`** ***copulas***. Just as in C, the entity that is assigned to the name can be the result of evaluating an expression.

There are a number of additional capabilities of J assignment that you can read about in the Dictionary. One that has no counterpart in C is that the name being assigned can itself be a variable, i. e. you can calculate the name that you want to assign the value to.

The value assigned can be a noun (**object**), verb (**function**), adverb, or conjunction; the name then becomes whatever part of speech was assigned to it (even if it was previously defined as a different part of speech!). For example,

```
n =: 5
```

creates a noun, and

```
v =: verb define
x. + y.
)
```

creates a verb (more below).

Note: the J Dictionary uses the terms 'local' and 'global' instead of 'private' and 'public'. I think 'private' and 'public' are more accurate terms, because there is another dimension to name scope in J, using the J notions ***locale*** and ***path***, that causes public variables to be visible only in certain entities. It will be a long time before we learn about locales; until then, public names will be global.

Order of Evaluation

Forget the table of operator precedence! All J verbs (**functions** and **operators**) have the same priority and associate **right-to-left**. For example, `a * b + c` is equivalent to `a * (b + c)`, not `(a * b) + c`. Use care when copying mathematical formulas. Note that the negative sign _ is a part of the number, not a verb. `_5 + _4` is `_9`, while `-5 + -4` is `_1`.

The executable bits of a sentence (**statement**) are called ***fragments*** (**subexpressions**). A verb with its operand(s) is a fragment, as is a copula with its name and value. We will meet other types of fragment later. ***Execution*** of a sentence consists of the right-to-left execution of its fragments, with the result of each fragment's execution replacing the fragment and being passed as an operand into the next fragment. The result of the last execution becomes the result of the sentence. This result is usually a noun but it can be any of the primary parts of speech. As an example, execution of the sentence

```
   a =. 3 + b =. 4 * 5
```

consists of execution of the following fragments: `4 * 5` with result `20`; `b =. 20` with result `20`; `3 + 20` with result `23`; `a =. 23` with result `23` . The names `a` and `b` are assigned when the assignment fragments are executed.

What a verb (function) looks like

As we saw, a J verb (**function**) is defined by lines that look like:

```
name =: verb define
J sentences here
)
```

The result of the `verb define` is a verb, and normally you will assign the result to a name so you can execute the verb by name when you need it. Subsequent lines, starting with the one after `verb define` and ending before the next line containing only the word `')'`, are read and saved as the text of the verb (heaven help you if you leave out the `)`!). The verb is not 'compiled'—only the most rudimentary syntax checking is performed; the text is saved and will be interpreted when the verb is executed.

Each line of the verb is a ***sentence*** (**statement**). The result of the last sentence executed becomes the result of the whole verb (this is not precisely true but it's close enough for now—details will be revealed in 'Control Structures').

Since a J verb has only one or two operands, there is no need for you to provide a list of parameter names as you do in a function definition in C; instead, J names them for you. At the start of a verb's execution, the private name `y.` is initialized with the value of the right operand of the verb. If the verb is dyadic, the private name `x.` is initialized with the value of the left operand. Many programmers like to start their verbs by assigning these values to more descriptive names.

If your verb is going to define only a monadic or dyadic form, you should use `monad define` or `dyad define` instead of `verb define` . If you are going to define both valences, the way to do so is:

```
name =: verb define
monadic case here
:
dyadic case here
)
```

where a line with the single word `:` separates the two cases. If you use `verb define` and don't have the `:`, the verb will be monadic.

If your verb is only one line long (not at all unusual in J!) you can define it all in one line by using the appropriate one of the forms

```
name =: monad : 'text of verb'
name =: dyad : 'text of verb'
```

Running a J program

No compiling. No linking. No makefiles. No debugger. You simply type J sentences and the interpreter executes them and displays any result. At the very simplest, you can use it as a desk calculator:

```
   22 + 55
77
```

J prints 3 spaces as a prompt, so when you scroll through the log of a session, your input will be indented 3 spaces while J's typeout will be unindented. The result of a sentence typed on the keyboard is displayed, except that to avoid excessive typeout nothing is displayed if the last fragment executed in the sentence is an assignment. If you are at the keyboard while you are reading this book, you can type the examples and see the responses, or experiment on your own.

Here is a simple program to add twice the left argument to three times the right argument:

```
   add2x3y =: dyad : '(2 * x.) + 3 * y.'
```

We can run this program by giving it operands:

```
   1 2 3 add2x3y 4 5 6
14 19 24
```

Instead of simply displaying the result, we can assign it to a noun:

```
   a =: 1 2 3 add2x3y 4 5 6
```

We can inspect the value assigned to the noun by typing the name of the noun:

```
   a
14 19 24
```

We can use the noun in an expression:

```
   2 * a
28 38 48
```

We can create a new verb that operates on the noun:

```
   twicea =: monad : '2 * a'
   twicea ''
28 38 48
```

Notice the `''` after the invocation of `twicea`. Remember, to invoke a verb you must give it an operand, even if the verb doesn't use an operand. `''` is just an empty string; `0` or any other value would work too. If you leave out the operand, J will show you the value of the name; since `twicea` is a verb, its value is the definition of the verb:

```
   twicea
3 : '2*a'
```

Of course, in any practical application you will need to have most of your programs in a library so you can quickly make them all available to J. J calls these libraries ***scripts*** (filename extension '.ijs') and runs them with the `load` verb, for example:

```
load 'system\packages\misc\jforc.ijs'
```

`load` reads lines from the script and executes them. These lines will normally be all the verb and noun definitions your application needs, possibly including `load` commands for other scripts. A script may end with a line executing one of the verbs it defined, thereby launching the application; or, it may end after defining names, leaving you in control at the keyboard to type sentences for J to execute.

Note: Names defined by private assignment (using `=.`) when a script is loaded are not available outside the script. If you want to define names for use elsewhere, make sure you use `=:` for your assignments within a script.

If you are used to debugging with Visual C++™ or the like, you will find the environment less glitzy and more friendly. If you want to change a verb (**function**), you simply edit the script, using the editor of your choice (I use the built-in editor provided with J), and rerun it. The verb will be updated, but all defined nouns (**objects**) will be unchanged. Even if you are running a large application—yea, even if the application is in the middle of reading from an asynchronous socket—you can change the program, without recompiling, relinking, or reinitializing. If you'd like to add some debugging code while the system is running, go right ahead. This easy interaction with an executing program is one of the great benefits of programming in J.

Interrupting Execution

If a J verb is taking too long to run, press the BREAK key (Ctrl+BREAK, in Windows) to return control to the keyboard.

Errors

When a sentence contains an error, J stops and displays the sentence along with a terse error message. Refer to the chapter on Error Messages for explanation of the error.

The Execution Window; Script Windows

When J starts it displays its ***execution window***. The title of the execution window ends with the characters `'.ijx'`. The only way to have a sentence executed is to have the sentence sent to the execution window. The simplest way to do that is by typing the sentence into the execution window, as we have been doing in the examples so far.

The execution window is an edit window and a session log as well as a place to type sentences for execution. If you put the cursor on some line other than the last and press ENTER, the line you were on will be copied to the bottom of the session log as if you had typed it for execution. You can then edit the line before pressing ENTER again to execute it.

For convenience in editing, you may create other windows which will be ***script windows***. Usually these windows will contain J scripts that you are working on: the editor that manages the script windows is familiar with the syntax of J. You create a script window by clicking File on the Menu Bar and then selecting New ijs, Open, or Recent.

Sentences that you type into a script window are not automatically executed by J; you must copy them into the execution window to have them executed. You can use the script-window editor to send lines from a script to the execution window: click Run on the Menu Bar and then File, Selection, or Window as appropriate.

To run a selection of lines from a script window, be sure to use Run|Selection rather than cut-and-paste. If you paste a number of lines into the execution window, only the last one will be executed.

Names Defined at Startup

When J starts, a number of useful names are defined. Rather than discuss them all, I will show you how they come to be defined so you can study them when you need to.

When J starts, it executes the script ***`J-directory`*****`\system\extras\config\profile.ijs`** which then executes the script ***`J-directory`*****`\system\extras\util\boot.ijs`** . **`boot.ijs`** in turn executes a series of scripts in ***`J-directory`*****`\system\main`** which define the starting environment. Look at these scripts to see what they define.

If you want to add your own initial definitions, do so either by adding commands at the end of **`profile.ijs`** or by creating your own startup script in ***`J-directory`*****`\system\extras\config\startup.ijs`** .

Step-By-Step Learning: Labs

The ***Labs*** are interactive demos describing various topics in J. To run the lab for printf, start a J session, on the menu bar select Studio|Labs…, then select the lab you are interested in, then press 'Run'. The lab provides explanatory text interspersed with examples executed in your J session which you are free to experiment with as you step through the lab.

I recommend that every now and again you tarry a while among the labs, running whichever ones seem interesting. Much of the description of the J system can be found only there.

J Documentation

The J documentation is available online. Pressing F1 brings up the *Vocabulary* page, from which you can quickly go to the Dictionary's description of each J primitive. At the top of each page of documentation are links to the manuals distributed with J: these are:

The *Index* to all documentation;

The *User Manual* which describes components of J that are not in the language itself, including system libraries and external interfaces;

The *J Primer*, an introduction to J;

J Phrases, a collection of useful fragments of J (you will need to finish this book before trying to use *J Phrases*);

The *J Dictionary*, the official definition of the language;

Release Notes for all releases of J;

A description of foreign conjunctions (*!:*);

A description of the operands to the *wd* verb (Windows interface).

Getting Help

Your first step in learning J should be to sign up for the J Forum at www.jsoftware.com. A great many experienced J users monitor messages sent to the Forum and are willing to answer your questions on J, from the trivial to the profound.

---

3. A First Look At J Programs

Before we get into learning the details of J, let's look at a couple of realistic, if simple, problems, comparing solutions in C to solutions in J. The J code will be utterly incomprehensible to you, but we will nevertheless be able to see some of the differences between J programs and C programs. If you stick with me through this book, you will be able to come back at the end and understand the J code presented here.

Average Daily Balance

Here is a program a bank might use. It calculates some information on accounts given the transactions that were performed during a month. We are given two files, each one containing numbers in lines ended by (CR,LF) and numeric fields separated by TAB characters (they could come from spreadsheets). Each line in the Accounts file contains an account number followed by the balance in the account at the beginning of the month. Each line in the Journal file contains an account number, the day of the month for a transaction, and the amount of the transaction (positive if money goes into the account, negative if money goes out). The records in the Journal file are in order of date, but not in order of account. We are to match each journal entry with its account, and print a line for each account giving the starting balance, ending balance, and average daily balance (which is the average of each day's closing balance). The number of days in the month is an input to the program, as are the filenames of the two files.

I will offer C code and J code to solve this problem. To keep things simple, I am not going to deal with file-I/O errors, or data with invalid format, or account numbers in the Journal that don't match anything in the Accounts file.

C code to perform this function might look like this:

```
#include <stdio.h>
#define MAXACCT 500
// Program to process journal and account files, printing
// start/end/avg balance.  Parameters are # days in current
// month, filename of Accounts file, filename of Journal file
void acctprocess(int daysinmo, char * acctfn, char *jourfn)
{
    FILE fid;
    int nacct, acctx;
    float acctno, openbal, xactnday, xactnamt
    struct {
        float ano;          // account number
        float openbal;    // opening balance
        float prevday;    // day number of last activity
        float currbal;     // balance after last activity
        float weightbal; // weighted balance: sum of closing balances
    } acct[MAXACCT];

    // Read initial balances; set day to start-of-month, sum of balances
to 0
    fid = fopen(acctfn);
    for(nacct = 0;2 == fscanf(fid,"%f%f",acctno,openbal) {
        acct[nacct].ano = acctno;
        acct[nacct].openbal = openbal;
        acct[nacct].prevday = 1;
        acct[nacct].currbal = openbal;
        acct[nacct].weightbal = 0;
        ++nacct;
    }
    fclose(acctfn);

    // Process the journal: for each record, look up the account
    // structure; add closing-balance values for any days that
    // ended before this journal record; update the balance
    fid = fopen(jourfn);
    while(3 == fscanf(fid,"%f%f%f",acctno,xactnday,xactnamt) {
        for(acctx = 0;acct[acctx].ano != acctno;++acctx);
        acct[nacct].weightbal +=
            acct[nacct].currbal * (xactnday - acct[nacct].prevday);
        acct[nacct].currbal += xactnamt;
        acct[nacct].prevday = xactnday;
    }
    // Go through the accounts.  Close the month by adding
    // closing-balance values applicable to the final balance;
    // produce output record
    for(acctx = 0;acctx < nacct;++acctx) {
        acct[nacct].weightbal +=
            acct[nacct].currbal * (daysinmo - acct[nacct].prevday);
        printf("Account %d: Opening %d, closing %d, avg %d\n",
            acct[acctx].ano, acct[acctx].openbal, acct[acctx].currbal,
             acct[acctx].weightbal/daysinmo);
    }
    fclose(fid);
}
```

The corresponding J program would look like this:

```
NB. Verb to convert TAB-delimited file into numeric
array
rdtabfile =: (0&".;.2@:(TAB&,)@:}:);._2) @ ReadFile @<

NB. Verb to process journal and account files
NB. y. is (# days in current month);(Account filename);
NB.   (Journal filename)
acctprocess =: monad define
'ndays acctfn jourfn' =: y.
NB. Read files
'acctano openbal' =. |: rdtabfile acctfn
'jourano jourday jouramt' =. |: rdtabfile jourfn

NB. Verb: given list of days y., return # days that
NB. each balance is a day's closing balance
wt =. monad : '(-~ 1&(|.!.(>:ndays))) 0{"1 y.'
NB. Verb: given an Account entry followed by the Journal
NB. entries for the account, produce (closing balance),
NB.  (average daily balance)
ab =. monad : '(wt y.)({:@] , (%&ndays)@(+/)@:*)+/\1{"1
y.'

NB. Create (closing balance),(average daily balance) for
NB. each account.  Assign the start-of-month day (1) to
```

```
the
NB. opening balance
cavg =. (acctano,jourano) ab/.(1,.openbal),jourday,.
jouramt

NB. Format and print all results
s =. 'Account %d: Opening %d, closing %d, avg %d\n'
s&printf"1 acctano ,. openbal ,. cavg
''
)
```

Let's compare the two versions. The first thing we notice is that the J code is mostly commentary (beginning with `NB.`). The actual processing is done in 3 lines that read the files, 3 lines to perform the computation of closing and average balance, and 2 lines to print the results. J expresses the algorithm much more briefly.

The next thing we notice is that there seems to be nothing in the J code that is looping over the journal records and the accounts. The commentary *says* 'create balances for each account' and 'produce average daily balance for an account', tasks that clearly require loops, and yet there is nothing resembling loop indexes. This is one of the miracles of J: loops are implied; in C terminology, they are **expressions** rather than **statements**, and so they can be assembled easily into single lines of code that replace many nested loops. We will be spending a lot of time learning how to do this.

We also note that there is nothing in the J code corresponding to the **#define MAXACCT 500** in the C. This is one of the things that makes programming in J so pleasant: you don't have to worry about allocating storage, or freeing it, or wondering how long is long enough for a character-string variable, or how big to make an array. Here, even though we don't know how many accounts there are until we have read the entire Accounts file, we simply read the file, split it into lines and numbers, and let the interpreter allocate as much storage as it needs to hold the resulting array.

The last thing to see, and perhaps the most important, is that *the C version is just a toy program*. It searches through the Accounts information for every record in the Journal file. We can test it with a small dataset and verify that it works, but if we scale it up to 10,000 accounts and 1,000,000 journal entries, we are going to be disappointed in the performance, because its execution time will be proportional to $A*J$ where A is the number of accounts and J the number of journal entries. It is every programmer's dread: a function that will have to be rewritten when the going gets tough.

The J version, in contrast, will have execution time proportional to *(A+J)*log(A+J)*. We did nothing meritorious to achieve this better behavior; we simply expressed our desired result and let the interpreter pick an implementation. Because we 'think big'—we treat the entire Journal and Accounts files as units—we give the interpreter great latitude in picking a good algorithm. In many cases the interpreter makes better decisions than we could hope to, because it looks at the characteristics of the data before it decides on its algorithm. For example, when we sort an array, the interpreter will use a very fast method if the range of numbers to be sorted is fairly small, where 'fairly small' depends on the number of items to be sorted. The interpreter takes great care in its implementation of its primitives, greater care than we can normally afford in our own C coding. In our example, it will use a high-speed method for matching journal entries with accounts.

Calculating Chebyshev Coefficients

This algorithm for calculating coefficients of the Chebyshev approximation of a function is taken verbatim from Numerical Recipes in C. I have translated it into J just so you can see how compact the J representation of an algorithm can be. Again, the J code will be gobbledygook for now, but it's *concentrated* gobbledygook.

```
// Program to calculate Chebyshev coefficients
// Code taken from Numerical Recipes in C 1/e
#include <math.h>
#define PI 3.141592653589793
void chebft(a,b,c,n,func)
float a,b,c[ ];
float (*func)();
int n;
{
    int k,j;
    float fac,bpa,bma,f[300];

    bma = 0.5 * (b-a)
    bpa = 0.5 * (b+a)
    for(k = 0;k<n;k++) {
        float y = cos(PI*(k+0.5)/n);
        f[k] = (*func)(y*bma+bpa);
    }
    fac = 2.0/n;
    for (j = 0;j<n;j++) {
        double sum = 0.0;
        for(k = 0;k<n;k++)
            sum += f[k] * cos(PI*j*(k+0.5)/n);
        c[j] = fac*sum;
    }
}
```

J version:

```
chebft =: adverb define
:
f =. u. 0.5 * (+/y.) - (-/y.) * 2 o. o. (0.5 + i. x.) %
x.
(2 % x.) * +/ f * 2 o. o. (0.5 + i. x.) *"0 1 (i. x.) %
x.
)
```

<< >> Contents Help

4. Declarations

J has no declarations. Good riddance! No more will you have to warn the computer of all the names you intend to use, and their types and sizes. No more will your program crash if you step outside an array bound. You specify the calculations you want to perform; if, along the way, you want to assign a result to a name, J will allocate enough space for the data. It will free the space when the name is no longer needed.

Seasoned C programmers have learned to use declarations to create a web of type-checking, making sure that objects pointed to are of the expected type. This is an example of making a virtue of necessity. Since J solves the problem much more directly—by not having pointers at all—you will soon lose your uneasiness with weak typing.

Arrays

But, you ask, without declarations, how does the computer know that a name represents an array? For that matter, how do *I* know that a name represents an array?

The answer affords a first glimpse of the power of J. Every J verb, whether a primitive (**operator**) or a user-written verb (**function**), accepts arguments that can be arrays, even multidimensional arrays. How is this possible? Like this: Suppose you write a verb that works with 2-dimensional arrays. Part of your verb definition will indicate that fact. If your verb is executed with an argument that is a 3-dimensional array, J will automatically split the 3-dimensional array into a sequence of 2-dimensional arrays, call your verb, and put the pieces back together into an array of results.

We will very soon go into this procedure in great detail. For now, you should learn the vocabulary J uses to deal with arrays.

What C calls an ***n*****-dimensional array of rank *i*×*j*×…×*k*** is in J an ***array*** of ***rank*** *n* with ***axes*** of length *i*,*j*,…,*k*.

Every noun (**variable** or **object**) has a ***shape*** which is the array (of rank 1) made by concatenating the lengths of all its axes. For example, if **q** is the array corresponding to the C array defined by the declaration

```
int q[4][5][6];
```

its shape is the array **4 5 6** . As you can see, the number of items in the shape is exactly the rank.

Note: a sequence of numbers written with no intervening punctuation defines a numeric array of rank 1 (i. e. a list). You may have to use parentheses if you have adjacent numbers that you don't want to have made into a list.

Unlike in C, an array in J may have one or more axes of length 0. Such an array has no atoms, but its rank is still the number of its axes.

A single number or character is called an ***atom*** (**object of basic type**) which is said to have the ***type*** numeric or character as appropriate. (Actually, there are types other than number and character, including a type that resembles a **structure**, but we won't get to them for a while). An atom is also called a ***scalar***. An atom is defined to have rank 0; therefore, its shape is an array with 0 items, i. e. an empty array of rank 1.

Just as in C, every atom of an array must have the same type.

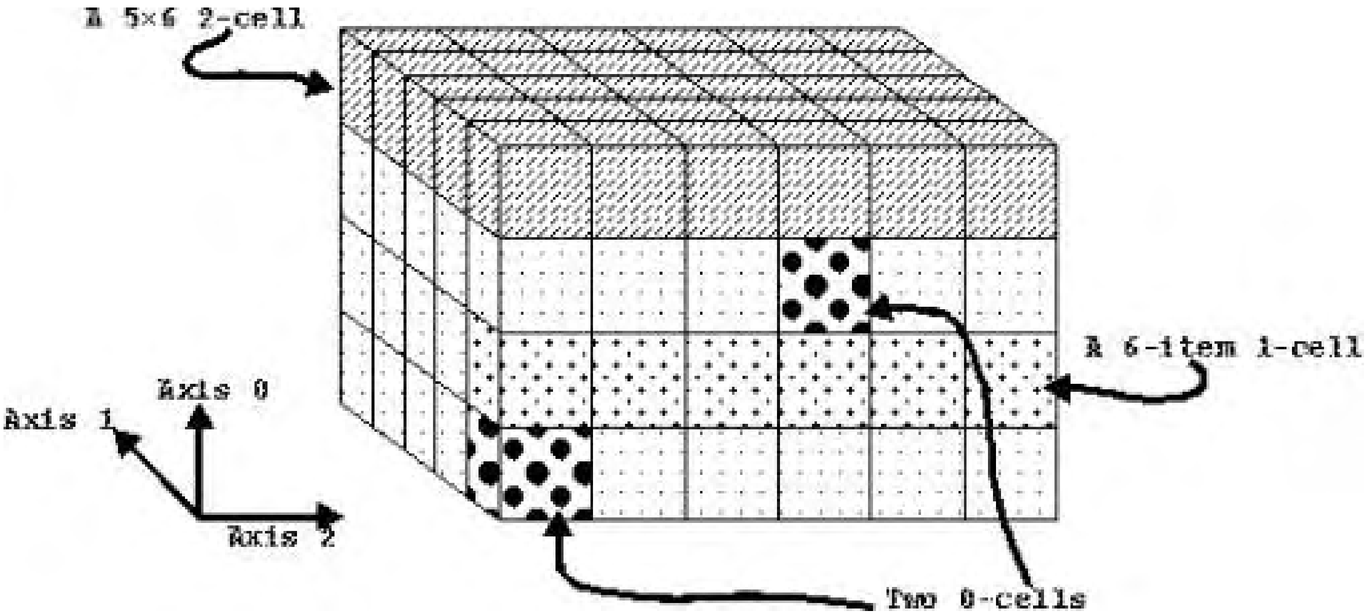
Cells

Because the execution of every J verb involves breaking the argument into pieces, presenting the pieces to the verb, and assembling results, J has a vocabulary for describing these operations.

A rank-3 array of shape **4 5**, 6 such as the one defined in C by the declaration

```
int q[4][5][6];
```

can be thought of as an array of 4 elements, each with rank 2 and shape **5 6**, or as a 4×5 array of elements, each with rank 1 and shape **6**, or as a 4×5×6 array of rank-0 atoms. The term ***cell*** is used to indicate the rank of the elements that will be operated on. A ***0-cell*** is an atom, a ***1-cell*** is an element of rank 1, a ***2-cell*** is an element of rank 2, and so on.



Once you have picked a cell size, you can think of your noun as an array of cells; the shape of that array is called the ***frame*** of the noun relative to the chosen rank of cell. It follows that the frame, concatenated with the shape of the cells, will be equal to the shape of the noun. The frame itself (like all shapes) is an array of rank 1.

The diagram illustrates cells of different ranks. Note that the twenty 6-atom 1-cells are arranged in a 4×5 array; this is the meaning of the frame of the 1-cells. The four 5×6 2-cells are arranged as a vector of 2-cells; this is the meaning of their frame.

A selected cell is analogous to the subarray selected by indexing in C. Using **q** as defined above, in C **q[3]** is a 5×6 array (i. e. a 2-cell); **q[1][0]** is a 6-element vector (i. e. a 1-cell); **q[2][0][3]** is a scalar (0-cell).

The noun **q** we have been using as an example can be thought of in any of the following 4 ways:

	Frame		Cells	
as an array of	Length	Value	Rank	Shape
0-cells	3	**4 5 6**	0	(empty)
1-cells	2	**4 5**	1	**6**
2-cells	1	**4**	2	**5 6**
3-cells	0	(empty)	3	**4 5 6**

Choosing Axis Order

Because J verbs operate on cells of nouns, you should choose an order of axes that makes the cells meaningful for your application. Referring to the figure, we can see that the groups of items that fall into horizontal strips (1-cells) or horizontal slabs (2-cells) will be easy to operate on individually. Vertical strips or slabs will not correspond to cells and so will not be accessible individually; to work on them we may have to reorder the axes of the noun to make them correspond to cells. Such reordering of axes is easy in J but it can often be avoided by ordering the axes properly in the first place.

Negative Cell-Rank; Items

In some cases, you may know the length of the frame and want to define the cells to have whatever rank is left over (for example, you may have a noun with one cell per employee, but you may not know the rank of the cells). We say that you are selecting the cells of the noun relative to the given frame. Negative cell-ranks indicate this. A _1-cell has the shape corresponding to a frame of length 1 (**`5 6`** in our example), a _2-cell has the shape corresponding to a frame of length 2 (**`6`** in our example), and so on. If the specified frame is longer than the rank of the noun, the entire shape of the noun is used for the frame (and the cells are 0-cells, i. e. atoms). In our example, a _3-cell, a _4-cell, a _5-cell, etc., all refer to atoms. As an important case of this, the _1-cell of an atom is the atom itself.

_1-cells are so important in J that they are given the name ***items***. Our example has 4 items, each of shape **`5 6`** . An atom has one item, itself.

Remember: an atom has one item: itself.

The ***index*** of an item in an array is the sequence number of the item from the beginning of the array. The first item in an array has the index 0, just as in C.

Lists

When we choose to view an array as a collection of its items, we say that the array is a ***list*** of its items. In our example above, the items of the 4×5×6 array are 5×6 arrays, and we say that the whole array is a list of **`4`** items each with shape **`5 6`** (equivalently, we say that it is a list of four 5×6 arrays). So many of J's primitives operate on items of their operands that we will find ourselves usually thinking of an array as a list of its items.

When the word 'list' is used without any indication of what the list contains, the list is assumed to contain atoms. So, 'the list **`x`**' refers to an array of rank 1 (**one-dimensional array**). **`0 3 5`** is a numeric list.

Note that J's use of the term 'list' has nothing to do with linked lists such as you are familiar with, where an element in the list contains a pointer to other elements. Since J has no pointers at all, you will not need that meaning, and you can get used to calling rank-1 arrays 'lists'. A list can also be called a ***vector***.

Phrases To Memorize

An **array** is a **list** of its **items**.

An atom has one item, itself.

The **rank** of a noun is the **length** of its **shape**.

The **shape** of an **atom** is the **empty list**.

The **suffixes** of the **shape** of a noun give the shapes of its **cells**: the *k* trailing atoms of the shape of a noun give the shape of its *k*-cell.

The **frame** of a noun **with respect to *k*-cells** is the **shape** of the noun **with the last *k* atoms removed**.

Constant Lists

A character or numeric list can be created simply by including the list in a sentence. We have seen that a sequence of numbers separated by spaces is recognized as a single word representing the list. Similarly, a character or a character list can be represented directly by a quoted string. C distinguishes between single-character constants (such as `'a'`) and strings (such as `"abc"`), using single quotes for characters and double quotes for strings. J uses only single quotes for defining character constants (the `"` character is a primitive in its own right). If exactly one character is between the quotes, the value is an atom; if none or more than one, the result is a list.

Array-creating verbs

Now that we know how to talk about arrays, we might as well create a few and see what they look like. As mentioned earlier, every J verb can be used to create an array—there are no special 'declaration' verbs—but we will start with a couple that do little else. The J lines are taken from an interpreter session; you can type them into your own session and get the same results. The indented lines were typed into J, and the unindented ones are J's responses.

Dyad $ ($hape) and monad $ ($hape Of)

The verb dyad `$` is invoked as `x $ y` . The result of dyad `$` has the frame `x` relative to the rank of the items of `y`, and is made up of the items of `y`, repeated cyclically as needed. It follows that the shape of this result is `x` concatenated with the shape of an item of `y` .

We will have to work together on this. Confronted with a definition like that, you might: (a) decide that J must be a language for tax accountants, and give up; (b) decide the definition is Greek and go on, hoping it will make sense later; (c) try a few examples to get an idea for what the definition means; (d) read it over and over again until you understand it. I hope you will eschew (a) and (b), and settle for no less than full understanding. For my part, I will offer a few useful examples that you can compare against the definition. Not everything in J will be as abstract as this.

```
   5 $ 2
2 2 2 2 2
```

The simplest case, creating (and displaying) a list of 5 items, each of which is `2` . Let's see how this result matches the definition. `y` is a scalar, so it has one item, which is also a scalar. Therefore, the result has the shape `5` (`x` (i. e. `5`) concatenated with the shape of an item of `y`; the shape of a scalar item of `y` is the empty list; `5` concatenated with an empty list is a list with the single element `5`). The scalar is repeated to fill the 5 items of the result. J displays a 1-cell on a single line, as shown.

```
   5 $ 'ab'
ababa
```

Now `y` is a list, but its items are still scalars, with rank 0 and shape empty; so the result still has the shape `5` . The 5 items come from the items of `y`, cyclically.

We can distill the foregoing analysis above to the observation that **when `y` is an atom or a list, `x` specifies the shape of `x $ y`** .

```
   4 4 $ 'There is a tide in the affairs of men'
Ther
e is
 a t
ide
```

The items of `y` are still scalars, with rank 0 and shape empty; the result has the shape `4 4` . The 16 items come from the items of `y`, cyclically. Not all items of `y` are used. J displays a rank-2 array as a sequence of lines, one for each 1-cell.

```
   0 $ 2

```

(The display is a single blank line) Just like `5 $ 2`, but the resulting list has 0 items, i. e. it is an empty list.

```
   1 $ 2
2
```

Similarly, a 1-item list.

```
   (0 $ 2) $ 2
2
```

Here `(0 $ 2)` produces an empty list, as we saw above, and that is the `x` to the second `$` . The items of `y` are still scalars, so the result has shape empty (an empty list concatenated with an empty list), i. e. it is a scalar.

The displays of a scalar and a 1-item list are identical. Does that mean that a scalar is the same thing as a 1-item list? No. I mean **no.** **NO!** They are not (I say this with the same resignation as when I tell my kids not to rollerblade too fast down our street, knowing that only painful experience will drive the message home). How can you tell them apart? What we need is a way to see the shape of a noun.

That way is monad `$` . The result of `$ y` is the shape of `y` (always a numeric list). For example:

```
   $ 1 2 3 4
4
```

A 4-item list has the shape `4` . Did you forget that `1 2 3 4` is a single list rather than 4 separate numbers? You can ask the interpreter how it splits a line into words by using monad `;:` :

```
   ;: '$ 1 2 3 4'
+-+-------+
|$|1 2 3 4|
+-+-------+
```

The words are shown in boxes. The list `1 2 3 4` is recognized as a single word.

```
   $ 6

```

The shape of a scalar is a 0-length list, as we have seen.

```
   $ 1 $ 2
```

```
1
```

Remember, all sentences are executed right-to-left, so this is `$ (1 $ 2)`, which gives the shape of the 1-item list. When a verb can be invoked dyadically, it is, so the rightmost `$` is executed as a dyad, not as a monad.

```
   $ (0 $ 2) $ 2

```

Here, we get the shape of the scalar—an empty list.

```
   $ 'a'

```

A single character is an atom, whose shape is the empty list.

```
   $'abc'
3
```

A 3-item list, one item for each character.

```
   $ ''
0
```

`''` is the empty character string, which you will see a lot of. Because it is easy to type, it is the value most often used when an empty list of any type will do.

Executing monad `$` twice gives the rank: `$ $ y` is the rank of `y` (as a single-item list). I suggest you not read on until you understand why.

Resuming our inquiries into dyad `$`, we have

```
   2 5 $ 1 10
 1 10  1 10  1
10  1 10  1 10
```

Again `y` is a list, so the items of `y` are scalars. The shape of the result is `2 5`, and the items of `y` are repeated to fill that shape. Note that the corresponding atoms in each cell are aligned in the display.

```
   1 5 $ 1 10
1 10 1 10 1
```

Similarly, but now the result has shape `1 5`. This is not the same as a 5-item list, which has shape `5` . Again, monad $ shows the shape:

```
   $ 1 5 $ 1 10
1 5
```

When `y` is not a scalar or a list, its items are not scalars, and `x` does not give the shape of the result. Let us work through an example using the definition of `x $ y` :

```
   3 $ 1 5 $ 1 10
1 10 1 10 1
1 10 1 10 1
1 10 1 10 1
```

Remember, this is processed as `3 $ (1 5 $ 1 10)`. The parenthesized part produces an array of shape `1 5`; since this has rank 2, its items are its 1-cells, each with shape `5`. The shape of the overall result is `x` concatenated with the shape of an item of `y`, to wit `3 5`. This is populated with the cells of `y`, of which there is only 1.

```
   3 $ 2 5 $ 'There is a tide in the affairs of men'
There
 is a
There
```

You should be able to explain where each line came from, and you should note that **in general, `x` specifies the frame of `x $ y` with respect to items of `y`** . When `y` is a list or an atom, its items are atoms and `x` gives the entire shape of the result.

```
   2 2 $ 2 5 $ 1 10
 1 10  1 10  1
10  1 10  1 10

 1 10  1 10  1
10  1 10  1 10
   $ 2 2 $ 2 5 $ 1 10
2 2 5
```

Now the shape of the result is `2 2 5`, a rank-3 array. J displays the 2-cells with a blank line in between. Similarly, a rank-4 array is displayed as all the 3-cells with 2 blank lines in between, and so on for higher ranks.

We have seen that the display of a zero-length list is a single blank line: proper, as there is one list, and it has no items. The display of a rank-2 array with no items is different: here we have zero lists, so we should expect no lines at all. This is indeed what happens:

```
   0 0$0
```

(there is no blank line). This is the result you should produce if you want a function to display nothing.

Here are two exercises. Once you can explain each result, you will be well on your way to becoming a J programmer. What will each of these sentences produce (answer on the next page)?

```
   3 1 $ 2 5 $ 1 10
   2 3 $ 2 5 $ 1 10 15
```

Solutions:

```
   3 1 $ 2 5 $ 1 10
 1 10  1 10  1

10  1 10  1 10

 1 10  1 10  1
   2 3 $ 2 5 $ 1 10 15
 1 10 15  1 10
15  1 10 15  1
 1 10 15  1 10

15  1 10 15  1
 1 10 15  1 10
15  1 10 15  1
```

Monad # (Tally)

The result of `# y` is a scalar, the number of items in `y` . This is the first item in the list `$y`, except that if `y` is an atom, `$y` is empty but `#y` is `1` (because, remember, an atom has one item, itself). If `y` is a list, `#y` is the length of the list. Quiz question: What is the difference between the results of `$ 1 2 3` and `# 1 2 3`?

```
   $ 1 2 3
3
   # 1 2 3
3
```

Answer: the result of monad `$` is always a list, but the result of monad `#` is a scalar:

```
   $ $ 1 2 3
1
   $ # 1 2 3

```

`#$y`, like `$$y`, gives the rank of `y` . Since monad `#` produces a scalar rather than a list, `#$y` is usually preferred. Just remember that the length of the shape is the rank.

Monad `i.` (Integers)

Monad **`i.`** has infinite rank and creates an array. **`i. y`** produces the same result as **`y $`** ***`ints`***, where ***`ints`*** is the list of all nonnegative integers in order. Examples:

```
   i. 5
0 1 2 3 4
```

A list of 5 items; the items are ascending integers.

```
   i. 2 3
0 1 2
3 4 5
```

A rank-2 result.

```
   i. 2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
```

A rank-3 result.

```
   i. 0

```

A list of 0 items.

```
   i. _5
4 3 2 1 0
```

If the argument is negative, the absolute value is used for the shape, but the items run in reverse order.

<<   >>   Contents   Help

5. Loopless Code I—Verbs Have Rank

Most J programs contain no loops equivalent to **while** and **for** in C. J does contain `while.` and `for.` constructs, but they carry a performance penalty and are a wise choice only when the body of the loop is a time-consuming operation. You are just going to have to learn to learn to code without loops.

I think this is the most intimidating thing about learning J—more intimidating even than programs that look like a three-year-old with a particular fondness for periods and colons was set before the keyboard. You have developed a solid understanding of loops, and can hardly think of programming without using them. But J is a revolutionary language, and all that is solid melts into air: you will find that most of your loops disappear altogether, and the rest are replaced by small gestures to the interpreter indicating your intentions.

Come, let us see how it can be done. I promise, if you code in J for 6 months, you will no longer think in loops, and if you stay with it for 2 years, you will see that looping code was an artifact of early programming languages, ready to be displayed in museums along with vacuum tubes, delay lines, and punched cards. Remember, in the 1960s programmers laughed at the idea of programming without **goto**s!

You are not used to classifying loops according to their function, but I am going to do so as a way of introducting J's primitives. We will treat the subject of loopless iteration in 6 scattered chapters, showing how to replace different variants of loops:

1. Loops where each iteration of the loop performs the same operation on different data;
2. Loops that apply an operation between all the items of the array, for example finding the largest item;
3. Loops where the operation to be performed on each cell is different;
4. Loops that are applied to regularly-defined subsets of the data;
5. Loops that are applied to subsets of the data defined irregularly;
6. Loops that accumulate information between iterations of the loop.

The simplest case is the most important, and we start with a few experiments.

Examples of Implicit Loops

```
   2 + 3 4 5
5 6 7
```

The verb dyad `+` is addition, and we have our first example of an implicit loop: the left argument `2` was added to each atom in the right argument.

```
   1 2 3 + 4 5 6
5 7 9
```

And look! If each operand is a list, the respective items are added. We wonder if the behavior of `2 + 3 4 5` was because items of the shorter operand are repeated cyclically:

```
   1 2 + 4 5 6
|length error
|   1 2    +4 5 6
```

Evidently not. A 'length error' means that the operands to `+` did not 'agree' (and you get an error if you try to add them). We will shortly understand exactly what this means.

```
   i. 2 3
0 1 2
3 4 5
```

A reminder of what monad `i.` does.

```
   0 100 + i. 2 3
  0   1   2
103 104 105
```

Whoa! The atoms of the left operand were applied to rows of the right operand. Interesting. This seems to be some kind of nested implicit loop.

Let's learn a couple of more verbs, monad `#.` and monad `#:` . Monad `#:` creates the binary representation of an integer (i. e. a list of `0`s and `1`s), and monad `#.` is its inverse, creating the integer from the binary representation. For the longest time I couldn't remember which was which, but at last I saw the mnemonic: the verb with the single dot (`#.`) creates an atom from a list; the verb with multiple dots (`#:`) creates a list from an atom:

```
   #: 5
1 0 1
   #. 1 0 1
5
```

Yes, they seem to perform as advertised. They can be applied to arrays:

```
   ]a =. #: 5 9
0 1 0 1
1 0 0 1
```

Look: the result is not a rank-1 list, but rather a rank-2 array, where each item has the binary representation of one operand value (and notice, an extra leading zero was added to the representation of 5). The little trick with `]a =.` will be explained later, but for now just think of `]a =.` as 'assign to `a` and display the result'. With `a` assigned, we have:

```
   #. a
```

```
5 9
```

This seems to be the desired result, but on reflection we are puzzled: how did the interpreter know to apply `#.` to each 1-cell rather than to each 0-cell? Contrast this result with the result of the verb monad `+:`, which means 'multiply by 2':

```
   +: a
0 2 0 2
2 0 0 2
```

Evidently the verbs themselves have some attribute that affects the rank of cell they are applied to. It's time for us to stop experimenting and learn what that attribute is.

The Concept of Verb Rank

Every verb has a ***rank***—the rank of the cells to which it is applied. If the rank of the verb's operand is smaller than the rank of the verb, the verb is applied to the entire operand and it is up to the author of the verb to ensure that it produces a meaningful result in that case.

Dyads have a rank for each operand, not necessarily the same.

A verb's rank can be infinite (_), in which case the verb is always applied to the operand in its entirety. In other words, if a verb has infinite rank for an operand, that operand is always processed as a single cell (having the rank of the operand).

If you don't know the rank of a verb, you don't know the verb. Using a verb of unknown rank is like wiring in a power-supply of unknown voltage—it will do something when you plug it in; it might even work; but if the voltage is wrong it will destroy what it's connected to. Avoid embarrassment! Know the rank of the verbs you use.

The definition page of each J verb gives the ranks of the verbs defined on the page, right at the top of the page after the name of the verb. Since most pages define both a monad and a dyad, you will usually find 3 numbers: the first is the rank of the monad, the other two are the left and right rank of the dyad. For example, click up the page for `#:` and you will see

```
#:  _ 1 0
```

which means that monad `#:` has infinite rank, while dyad `#:` has left rank 1 and right rank 0.

For any verb, including user-written verbs, you can ask the interpreter the rank by typing ***`verbname`*** `b. 0` :

```
   #: b. 0
_ 1 0
```

Verb Execution—How Rank Is Used (Monads)

The implicit looping in J results from the interplay of verb rank and noun rank. For monads, it goes like this:

1. Figure out the rank ***r*** of the cells that will be operated on; this will be the **smaller** of rank of the verb and the rank of the operand. This rule applies even if the verb has

infinite rank: *r* will be the rank of the operand, which is another way of saying that the verb applies to the operand in its entirety.

2. Find the frame ***f*** of the operand with respect to cells of rank *r*.

3. Think of the operand as an array with shape *f* whose items are cells of rank *r*. Apply the verb to each *r*-cell, replacing each cell with the result of the verb. Obviously, this will yield an array of shape *f* whose items have the shape of the result of applying the verb to an *r*-cell.

Let's look at some simple examples:

```
   i. 2 2
0 1
2 3
```

This will be the right operand.

```
   +: i. 2 2
0 2
4 6
```

The steps to get this result are:

<table>
<tr><td colspan="2">The verb rank is 0 and the noun rank is 2, so we will be applying the verb to 0-cells. The frame <i>f</i> is <code>2 2</code></td></tr>
<tr><td>Think of the operand as a 2×2 array of 0-cells:</td><td><table><tr><td>0</td><td>1</td></tr><tr><td>2</td><td>3</td></tr></table></td></tr>
<tr><td>The verb is applied to each cell:</td><td><table><tr><td>0</td><td>2</td></tr><tr><td>4</td><td>6</td></tr></table></td></tr>
<tr><td>Since each result is an atom, i. e. a 0-cell, the result is a 2×2 array of 0-cells, i. e. an array of shape <code>2 2</code></td><td><code>0 2</code>
<code>4 6</code></td></tr>
</table>

Figure 1. Execution of +: i. 2 2

Another example:

```
   ]a =. 2 2 4 $ 0 0 1 1  0 0 0 1  0 1 0 0  0 0 1 0
0 0 1 1
0 0 0 1

0 1 0 0
0 0 1 0
```

This is a rank-3 array.

```
   #. a
3 1
4 2
```

<table>
<tr><td colspan="2">The verb rank is 1 and the noun rank is 3, so we will be applying the verb to 1-cells. The frame <i>f</i> is <b><code>2 2</code></b></td></tr>
<tr><td>Think of the operand as a 2×2 array of 1-cells:</td><td><table><tr><td>0 0 1 1</td><td>0 0 0 1</td></tr><tr><td>0 1 0 0</td><td>0 0 1 0</td></tr></table></td></tr>
<tr><td>The verb is applied to each cell:</td><td><table><tr><td>3</td><td>1</td></tr><tr><td>4</td><td>2</td></tr></table></td></tr>
<tr><td>Since each result is an atom, i. e. a 0-cell, the result is a 2×2 array of 0-cells, i. e. an array of shape <b><code>2 2</code></b></td><td><b><code>3 1</code></b>
<b><code>4 2</code></b></td></tr>
</table>

Figure 2. Execution of #. 2 2 4 $ 0 0 1 1 0 0 0 1 0 1 0 0 0 0 1 0

Controlling Verb Execution By Specifying a Rank

The implicit loops we have used so far are interesting, but they are not powerful enough for our mission of replacing all explicit loops. To understand the deficiency and its remedy, consider the new verb monad **`+/`**, which creates the total of the items of its operand (just think of it as 'monad **`SumItems`**'):

```
   +/ 1 2 3
6
```

The result was **`1 + 2 + 3`**, as expected.

```
   i. 2 3
0 1 2
3 4 5
   +/ i. 2 3
3 5 7
```

The result was **`0 1 2 + 3 4 5`**, as expected (remember that the *items* are added, and the items of **`i. 2 3`** are 1-cells). Adding together a pair of 1-cells adds the respective atoms, as we will soon learn.

This application of monad **`+/`** to a rank-2 array corresponds to the C code fragment:

```
   for(j = 0;j<3;++j)sum[j] = 0;
```

```
for(i = 0;i<2;++i)
     for(j = 0;j<3;++j)sum[j] += array[i][j];
```

Suppose we wanted to add up the items of each row, as in the C code fragment

```
for(i = 0;i<2;++i) {
     sum[i] = 0;
     for(j = 0;j<3;++j)sum[i] += array[i][j];
}
```

to produce the result `3 12`? How can we do it in J? What we have learned so far is not enough, but if we had a way to make monad `+/` apply to 1-cells—if we could make monad `+/` have rank 1—our problem would be solved: the implicit looping would cause each **row** to be summed and the results collected.

You will not be surprised to learn that J does indeed provide a way to apply monad `+/` on 1-cells. That way is the ***rank conjunction*** `"` .

We will learn all about conjunctions later on—the syntax is a little different than for verbs—but for now, we'll try to understand this `"` . It's used like this:

```
   u"n
```

to produce a new verb that is **`u` applied to `n`-cells individually**. This is a simple idea, but its ramifications spread wide. As a first example:

```
   +/"1 i. 2 3
3 12
```

This is what we were looking for. It happened this way:

The verb rank is 1 and the noun rank is 2, so we will be applying the verb to 1-cells. The frame *f* is **2**	
Think of the operand as a list of 2 1-cells:	0 1 2 │ 3 4 5
The verb monad `+/` is applied to each cell:	3 │ 12
Since each result is an atom, i. e. a 0-cell, the result is a list of 2 0-cells, i. e. an array of shape **2**	**3 12**

Figure 3. Execution of +/"1 i. 2 3

Examples Of Verb Rank

Here are some more examples using a rank-3 array as data:

```
   i. 2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23

   +/"1 i. 2 3 4
6 22 38
54 70 86
```

<table>
<tr><td colspan="2">The verb rank is 1 and the noun rank is 3, so we will be applying the verb to 1-cells. The frame *f* is **2 3**</td></tr>
<tr><td>Think of the operand as a 2×3 array of 1-cells:</td><td><table><tr><td>0 1 2 3</td><td>4 5 6 7</td><td>8 9 10 11</td></tr><tr><td>12 13 14 15</td><td>16 17 18 19</td><td>20 21 22 23</td></tr></table></td></tr>
<tr><td>The verb monad **+/** is applied to each cell:</td><td><table><tr><td>6</td><td>22</td><td>38</td></tr><tr><td>54</td><td>70</td><td>86</td></tr></table></td></tr>
<tr><td>Since each result is an atom, i. e. a 0-cell, the result is a 2×3 array of 0-cells, i. e. an array of shape **2 3**</td><td>**6 22 38**
54 70 86</td></tr>
</table>

Figure 4. Execution of +/"1 i. 2 3 4

```
   +/"2 i. 2 3 4
12 15 18 21
48 51 54 57
```

<table>
<tr><td colspan="2">The verb rank is 2 and the noun rank is 3, so we will be applying the verb to 2-cells. The frame *f* is **2**</td></tr>
<tr><td>Think of the operand as a list of 2 2-cells:</td><td><table><tr><td>0 1 2 3
4 5 6 7
8 9 10 11</td><td>12 13 14 15
16 17 18 19
20 21 22 23</td></tr></table></td></tr>
</table>

The verb monad `+/` is applied to each cell. As we have learned, this sums the items, making each result a rank-1 list	12 15 18 21 │ 48 51 54 57

Since each result is a rank-1 list, i. e. a 1-cell, the result is a list of 2 1-cells, i. e. an array of shape `2 4`	`12 15 18 21`
	`48 51 54 57`

Figure 5. Execution of +/"2 i. 2 3 4

```
   +/"3 i. 2 3 4
12 14 16 18
20 22 24 26
28 30 32 34
```

The verb is applied to the single 3-cell. Its items, which are 2-cells, are added, leaving a single 2-cell as the result.

How about `i."0 (2 2 2)`—can you figure out what that will produce? (Notice I put parentheses around the numeric list `2 2 2` so that the rank `0` wouldn't be treated as part of the list)

The verb rank is 0 and the noun rank is 1, so we will be applying the verb to 0-cells. The frame *f* is `3`	
Think of the operand as a list of 3 0-cells (i. e. atoms):	2 │ 2 │ 2
The verb monad `i.` is applied to each cell:	0 1 │ 0 1 │ 0 1
Since each result is a list, i. e. a 1-cell, the result is a list of 3 1-cells each with shape `2`, i. e. an array of shape `3 2`	`0 1` `0 1` `0 1`

Figure 6. Execution of i."0 (2 2 2)

```
   i."0 (2 2 2)
0 1
0 1
0 1
```

If you worked through that, it might have occurred to you that the shape of each result cell depended on the value of the operand cell, and that if those cells had not been identical, there would be some rough edges showing when it came time at the end to join the dissimilar result cells together. If so, full marks to you! That can indeed happen. If it does, then just before the cells are joined together to make the final result, the interpreter will bulk up the smaller results to bring them up to the shape of the largest. First, if the ranks of the results are not identical, each result will have leading axes of length 1 added as needed to bring all the results up to the same rank (e. g. if one result has shape **`2 5`** and another has shape **`5`**, the second will be converted to shape **`1 5`**, leaving the data unchanged). Then, if the lengths of the axes are not identical, the interpreter will extend each axis to the maximum length found at that axis in any result: this requires adding atoms, called ***fills***, which are always **`0`** for numeric results and **`' '`** for literal results. Example:

```
   i."0 (0 1 2 3)
0 0 0   NB. original result was empty list; 3 fills added
0 0 0   NB. original result was 0; 2 fills added
0 1 0   NB. original result was 0 1; 1 fill added
0 1 2   NB. this was the longest result, no fill added
```

`fndisplay`—A Utility for Understanding Evaluation

J contains a script that we will use to expose the workings of evaluation. You define verbs which, instead of operating on their operands, accumulate character strings indicating what operations were being performed. This gives you a way of seeing the operations at different cells rather than just the results.

Start by loading the script:

```
   load 'system/packages/misc/fndisplay.ijs'
```

Then, select the type of display you want. We will be using

```
   setfnform 'J'
```

Then, give the names of the verbs you want to use. If you want to assign a rank, you may do so by appending **`"`*`r`*** to the name:

```
   defverbs 'SumItems plus"0'
```

Optionally, define any names you want to use as nouns. The value assigned to the noun is the noun's name:

```
   defnouns 'x y'
```

You are free to use other nouns in expressions, but they will be replaced by their values.

With these definitions, you can explore J's evaluations:

```
   x (plus) y
+--------+
|x plus y|
+--------+
```

The result of the evaluation is a description of the evaluation that was performed. The result is displayed in a box so that a sentence with multiple evaluations shows each in its proper place:

```
   SumItems"1 i. 2 3
+-------------+-------------+
|SumItems 0 1 2|SumItems 3 4 5|
+-------------+-------------+
```

Here we see that **`SumItems`** was applied twice, once on each 1-cell.

`fndisplay` cannot produce a valid result in cases where the rank of a verb is smaller than the rank of the result-cells of the preceding verb, because then the operation would be performed on part of a result cell, and the result cell is just a descriptive string which cannot be meaningfully subdivided.

In this book, if we give an example that starts with **`defverbs`** it is implied that **`load fndisplay`** and **`setfnform 'J'`** have been executed.

If you prefer to see the order of evaluation expressed in functional form like that used in C, you may issue **`setfnform 'math'`** before you execute your sentences:

```
   setfnform 'math'
   1 plus 2 plus y
+----------------+
|plus(1,plus(2,y))|
+----------------+
```

Negative Verb Rank

Recall that we defined the _1-cell of a noun ***n*** to be the cells with rank one less than the rank of *n*, and similarly for other negative ranks. If a verb is defined with negative rank ***r***, it means as usual that the verb will apply to *r*-cells if possible, but with *r* negative the rank of those *r*-cells will depend on the rank of the operand. After the first step of processing the verb, which decides what rank of cell the verb will be applied to, verbs with negative rank are processed just like verbs of positive rank.

```
   +/ "_1 i. 3
0 1 2
```

Here the _1-cells are atoms, so applying monad **`SumItems`** on each one has no effect.

```
   +/ "_1 i. 2 3
3 12
```

The operand has rank 2, so this expression totals the items in each 1-cell.

```
   +/ "_2 i. 2 2 3
 3 12
21 30
```

The operand has rank 3, so this totals the items in each 1-cell, leaving a 2×2 array of totals.

Verb Execution—How Rank Is Used (Dyads)

We are at last ready to understand the implicit looping that is performed when J processes a dyadic verb. Because a dyadic verb has two ranks (one for each operand), and these two ranks interact with each other as well as the ranks of the operands, you should not read further until you thoroughly understand what we have covered already.

We have learned that the rank conjunction `u"n` is used to specify the rank of a verb. Since each verb has the potential of being invoked monadically or dyadically, the rank conjunction must specify the ranks for both valences. This requires 3 ranks, since the monad has a single rank and the dyad a left and a right rank. The ranks `n` may comprise from 1 to 3 items: if 3 ranks are given they are, in order, the monad's rank, the dyad's left rank, and the dyad's right rank. If two ranks are given, the first is the dyad's left rank and the second is used for the dyad's right rank and the rank of the monad. If there is only one item in the list, it is used for all ranks. So, `v"0 1` has monad rank `1`, dyad left rank `0`, and dyad right rank `1` . As usual, J primitives themselves have the ranks shown in the Dictionary.

Processing of a dyad follows the same overall plan as for monads, except that with two operands there are two cell-sizes (call them ***lr*** and ***rr***) and two frames (call them ***lf*** and ***rf***). If the left and right frames are identical, the operation can be simply described: the *lr*-cells of the left operand match one-to-one with the *rr*-cells of the right operand; the dyad is applied to those matched pairs of cells, producing a result for each pair; those results are collected as an array with frame *lf*. We illustrate this case with an example:

```
   (i. 2 2) + i. 2 2
0 2
4 6
```

<table>
<tr><td colspan="3">The verb has left rank 0, and the left operand has rank 2, so the operation will be applied to 0-cells of the left operand. The verb has right rank 0, and the right operand has rank 2, so the operation will be applied to 0-cells of the right operand. The left frame is <b>2 2</b>, the right frame is <b>2 2 .</b></td></tr>
<tr><td>Think of the left operand as a 2×2 array of 0-cells, and the right operand as a 2×2 array of 0-cells:</td><td>0 1
2 3</td><td>0 1
2 3</td></tr>
<tr><td>The corresponding left and right operand cells are paired:</td><td colspan="2">0 0 | 1 1
2 2 | 3 3</td></tr>
</table>

The operation dyad **+** is performed on each pair of cells:	0 2 4 6

Since each result is an atom, and the frame is `2 2`, the result is an array with shape `2 2`	`0 2`
	`4 6`

Figure 7. Execution of (i. 2 2) + i. 2 2

Using **fndisplay**, we have

```
   load'system\packages\misc\fndisplay.ijs'
   setfnform 'J'
   defverbs 'plus"0'
   (i. 2 2) plus i. 2 2
+--------+--------+
|0 plus 0|1 plus 1|
+--------+--------+
|2 plus 2|3 plus 3|
+--------+--------+
```

As you can see, we were correct when we asserted that the sum of two cells of the same shape is taken by adding their respective items.

Concatenating Lists: Dyad , (Append)

For a second example we will introduce a new verb, dyad **,** (the verb is the comma character). **x , y** creates an array whose leading items are the items of **x** and whose trailing items are the items of **y**; in other words, it concatenates **x** and **y** .

```
   1 2 3 , 6
1 2 3 6
   1 2 3 , 1 2 3
1 2 3 1 2 3
   4 , 6
4 6
   1 2 3 , 0$6
1 2 3
   (i. 2 3) , (i. 3 3)
0 1 2
3 4 5
0 1 2
```

```
3 4 5
6 7 8
```

In the last example the items of `x` and `y` are 3-element lists, so `x , y` is a list of 3-element lists, containing the items of `x` followed by the items of `y` .

Dyad `,` has infinite rank, which means that it applies to its operands in their entirety and so its detailed operation is defined not by the implicit looping we have been learning, but instead by the definition of the verb in the Dictionary. The discussion given above describes the operation of dyad `,` when the operands have identically-shaped items (*items*, mind you—the *shapes* of `x` and `y` may differ, if one has more items than the other). In a later chapter we will learn about `x , y` when the items have different shapes; for now we will be dealing with operands that are scalars and lists, for both of which the items are scalars.

Now see if you can figure out what `(i. 3 3) ,"1 0 i. 3` will do before reading the explanation that follows:

<table>
<tr><td colspan="3">The verb (dyad <code>,"1 0</code>) has left rank 1, and the left operand has rank 2, so the operation will be applied to 1-cells of the left operand. The verb has right rank 0, and the right operand has rank 1, so the operation will be applied to 0-cells of the right operand. The left frame is <code>3</code>, the right frame is <code>3 .</code></td></tr>
<tr><td>Think of the left operand as a list of 3 1-cells, and the right operand as a list of 3 0-cells:</td><td>0 1 2 | 3 4 5 | 6 7 8</td><td>0 | 1 | 2</td></tr>
<tr><td>The corresponding left and right operand cells are paired:</td><td colspan="2">0 1 2 | 0   3 4 5 | 1   6 7 8 | 2</td></tr>
<tr><td>The operation dyad <code>,</code> is performed on each pair of cells:</td><td colspan="2">0 1 2 0 | 3 4 5 1 | 6 7 8 2</td></tr>
<tr><td>Since each result is a 1-cell of shape <code>4</code>, and the frame is <code>3</code>, the result is an array with shape <code>3 4</code></td><td colspan="2"><code>0 1 2 0</code>
<code>3 4 5 1</code>
<code>6 7 8 2</code></td></tr>
</table>

Figure 8. Execution of (i. 3 3) ,"1 0 i. 3

```
   defverbs 'comma'
```

```
    (i. 3 3) comma"1 0 i. 3
+-------------+-------------+-------------+
|0 1 2 comma 0|3 4 5 comma 1|6 7 8 comma 2|
+-------------+-------------+-------------+
   (i. 3 3) ,"1 0 i. 3
0 1 2 0
3 4 5 1
6 7 8 2
```

When Dyad Frames Differ: Operand Agreement

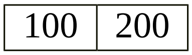
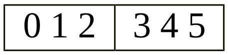
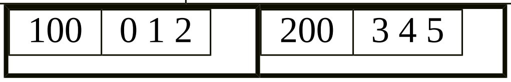
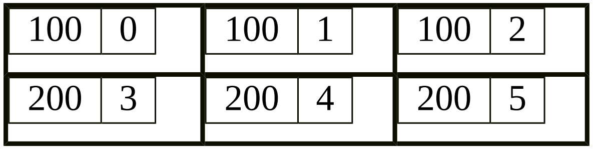
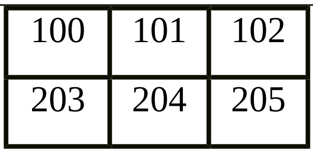
The processing of dyads has an extra step not present for monads, namely the pairing of corresponding cells of the left and right operands. As long as the frames *lf* and *rf* are the same, as in the examples so far, this is straightforward. If the frames are different, J may still be able to pair left and right cells, using another level of implicit looping, one that provides considerable additional programming power. The formal description that follows is not easy to follow—you might want to skim over it and read it in detail after you have studied the examples that follow.

J requires that one of the frames be a **prefix** of the other (if the frames are identical, each is a prefix of the other and all the following reduces to the simple case we have studied). The ***common frame cf*** is the part of the frames that is identical, namely the shorter of the two frames; its length is designated ***rcf***. If we look at the cells of the operands relative to this common frame (i. e. the (-*rcf*)-cells), we see that for the operand with the shorter frame, these cells are exactly the rank that will be operated on, while for the operand with the longer frame, each (-*rcf*)-cell contains multiple operand cells.

First, the (-*rcf*)-cells of the two operands are paired one-to-one (because they have the same frame), leaving each shorter-frame operand cell paired with a longer-frame (-*rcf*)-cell. Then, the longer-frame (-*rcf*)-cells are broken up into operand cells, with each operand cell being paired with a copy of the shorter-frame operand cell that was paired with the (-*rcf*)-cell (an equivalent statement is that the cells of the shorter-frame operand are replicated to match the surplus frame of the longer-frame operand). This completes the pairing of operand cells, and the operation is then performed on the paired operand cells, and collected using the longer frame. Maybe some examples will help.

```
   100 200 + i. 2 3
100 101 102
203 204 205
```

The verb (dyad **+**) has left rank 0, and the left operand has rank 1, so the operation will be applied to 0-cells of the left operand. The verb has right rank 0, and the right operand has rank 2, so the operation will be applied to 0-cells of the right operand. The left frame is **2**, the right frame is **2 3 .**

The common frame is **`2`**, with length 1, so think of each operand as a list of 2 _1-cells		
The _1-cells of the operands are paired:		
The longer-frame operand (the right one) is broken up into operand 0-cells, each being paired with a copy of the shorter-frame operand cell. Each paired _1-cell becomes a row of paired operand cells:	 100 0 100 1 100 2 200 3 200 4 200 5	
The operation dyad **`+`** is performed on each pair of cells:		

Since each result is an atom, and the longer frame is **`2 3`**, the result is an array with shape **`2 3`**	**`100 101 102`**
	`203 204 205`	

Figure 9. Execution of 100 200 + i. 2 3

```
   defverbs 'plus"0'
   100 200 plus i. 2 3
+----------+----------+----------+
|100 plus 0|100 plus 1|100 plus 2|
+----------+----------+----------+
|200 plus 3|200 plus 4|200 plus 5|
+----------+----------+----------+
```

The simplest and most common case of different-length frames is when the shorter frame is of zero length; in other words, when one of the operands has only one cell. In that case, the single cell is replicated to match every cell of the longer operand. An easy way to force an operand to be viewed as a single cell is to make the verb have infinite rank for that operand. This is not a special case—the behavior follows from the rules already given—but it's worth an example:

```
   'abc' ,"_ 0 'defg'
abcd
abce
abcf
abcg
```

<table>
<tr><td colspan="3">The verb (dyad <code>,"_ 0</code>) has left rank _, and the left operand has rank 1, so the operation will be applied to 1-cells of the left operand. The verb has right rank 0, and the right operand has rank 1, so the operation will be applied to 0-cells of the right operand. The left frame is (empty), the right frame is <b><code>4 .</code></b></td></tr>
<tr><td>The common frame is (empty), with length 0, so take each operand in its entirety:</td><td>abc</td><td>defg</td></tr>
<tr><td>The cells of the operands are paired:</td><td colspan="2">abc | defg</td></tr>
<tr><td>The longer-frame operand (the right one) is broken up into operand 0-cells, each being paired with a copy of the shorter-frame operand cell:</td><td colspan="2">abc | d
abc | e
abc | f
abc | g</td></tr>
<tr><td>The operation dyad <code>,</code> is performed on each pair of cells:</td><td colspan="2">abcd | abce | abcf | abcg</td></tr>
<tr><td>Since each result is a 1-cell with length <b><code>4</code></b>, and the longer frame is <b><code>4</code></b>, the result is an array with shape <b><code>4 4</code></b></td><td colspan="2"><b><code>abcd
abce
abcf
abcg</code></b></td></tr>
</table>

Figure 10. Execution of 'abc' ,"_ 0 'defg'

```
   defverbs 'comma'
   'abc' comma"_ 0 'defg'
+-----------+-----------+-----------+-----------+
|abc comma d|abc comma e|abc comma f|abc comma g|
+-----------+-----------+-----------+-----------+
```

You must thoroughly understand this example, where one operand has only one cell, because it occurs frequently. The handling of the general case of dissimilar frames is uncommon enough that you do not need to understand it perfectly right now—you'll know when you need it, and you can sweat out the solution the first few times. Here are a few observations that may help when that time comes:

It is always **entire cells of the operand with the shorter frame** that are replicated. A cell is never tampered with; nothing inside a cell will be replicated. And, it is not the entire shorter-frame operand that is replicated, but cells singly, to match the surplus frame of the other operand.

This fact, that single operand cells are replicated, is implied by the decision that the shorter frame must be a prefix of the longer frame: the single cell is the only unit that can be replicated, since the surplus frame is at the end of the frame rather than the beginning. Take a moment to see that this was a good design decision. Why should the following fail?

```
   1 2 3 + i. 2 3
|length error
|   1 2 3    +i.2 3
```

The 'length error' means that the operands do not ***agree***, because the frame-prefix rule is not met. Your first thought might be that adding a 3-item list to an array of 2 3-item lists should be something that a fancy language like J would do without complaining; if so, think more deeply. J does give you a way to add lists together—just tell J to apply the verb to lists:

```
   1 2 3 +"1 i. 2 3
1 3 5
4 6 8
```

Operands in which one shape is a suffix of the other, as in this example, are handled by making the verb have the rank of the lower-rank operand; that single operand cell will then be paired with all the cells of the other operand. By requiring dissimilar frames to match at the *beginning*, J gives you more control over implicit looping, because each different verb-rank causes different operand cells to be paired. If dissimilar frames matched at the end, the pairing of operand cells would be the same regardless of verb-rank.

Order of Execution in Implied Loops

Whenever a verb is applied to an operand whose rank is higher than the verb's rank, an implied loop is created, as we have discussed above. **The order in which the verb is applied to the cells is undefined**. The order used on one machine may not be that used on another one, and the ordering may not be predictable at all. If your verb has side effects, you must insure that they do not depend on the order of execution.

Current versions of the interpreter apply the verb to cells in order, but that may change in future releases.

A Mistake To Avoid

Do not fall into the error of thinking that `v"r` is '`v` with the rank changed to `r`'. It is not. Nothing can ever change the rank of the verb `v`—`v"r` is a *new verb* which has the rank `r`. This distinction will become important presently as we discuss nested loops. Consider the verb `v"1"2`,which is parsed as `(v"1)"2` . If `v"r` changed the rank of `v`, it would follow that `v"1"2` would be '`v` with the rank changed to 1 and then to 2', i. e. identical to `v"2` . But it is not: actually, `v"1"2` applies `v"1` on the 2-cells of the operand, while `v"2` applies `v` on those same cells—and we have seen that `v` and `v"1` are very different verbs:

```
   +/"1"2 i. 2 3 4
```

```
 6 22 38
54 70 86
   +/"2 i. 2 3 4
12 15 18 21
48 51 54 57
```

Summing the 2-cells (`+/"2`) is not the same as summing the 1-cells within each 2-cell (`+/"1"2`). Make sure you see why.

Ah, you may say, but `+/"1"2` is equivalent to `+/"1` . You are right for the monadic case, but not for the dyadic:

```
   (i. 3 4) +"1"2 i. 2 3 4
 0  2  4  6
 8 10 12 14
16 18 20 22

12 14 16 18
20 22 24 26
28 30 32 34
   (i. 3 4) +"1 i. 2 3 4
|length error
|   (i.3 4)    +"1 i.2 3 4
```

Dyad `+"1"2` is executed as `(+"1)"2`, i. e. it has rank 2. So, there is only one 2-cell of the left operand `i. 3 4`, and that cell is replicated to match the shape of the right operand. The operands then agree, and the 1-cells can be added. Trying to add the 1-cells directly with `+"1` fails, because the frames of the operands with respect to 1-cells do not agree.

The situation becomes even more complicated if the assigned left and right ranks are not the same. My advice to you is simple: remember that `u"r` is a **new verb** that executes `u` on `r`-cells.

---

6. Starting To Write In J

It's time to write some simple J programs. Writing code without loops will be a shock at first. Many accomplished C programmers give up because they find that writing every program is a struggle, and they remember how easy it was in C. I hope you will have more persistence. If you do, you will soon stop thinking in loops, translating each one into J; instead, you will think directly about operand cells, and the code will flow effortlessly.

In C, when you are going to operate on some array, say **x[3][4][5]**, you write the code from the outside in. You know you are going to need 3 nested loops to touch all the cells; you write the control structure for each loop (possibly after thinking a bit about the order of nesting); finally, you fill in code at whatever nesting level it fits. Even if all your work is in the innermost loop, you have to write all the enclosing layers just to be able to index the array. When I was writing in C, I made sure my output was measured in lines of code, so that I could call this 'productivity'.

In J, you grab the heart of the watermelon rather than munching your way in starting at the rind. You decide what rank of operand cell you are going to work on, and you write the verb to operate on a cell. You give the verb the rank of the cells it operates on, and then you don't care about the shape of the operand, because J's implicit looping will apply the verb to all the cells, no matter how many there are. A pleasant side effect of this way of coding is that the verbs you write can be applied to operands of any shape: write a verb to calculate the current value of a loan, and you can use that very verb to calculate the current value of all loans at a branch, or at all branches in the city, or all over the state.

We will write a number of J verbs starting from their C counterparts so you can see how you need to change your thinking. For these examples, we will imagine we are in the payroll department of a small consulting business, and we will answer certain questions concerning some arrays defined as follows:

> **empno[nemp]** (in J, just **empno**) - employee number for each member of the staff. The number of employees is **nemp**.
>
> **payrate[nemp]** - number of dollars per hour the employee is paid

billrate[nemp] - number of dollars per hour the customer is billed for the services of this employee

clientlist[nclients] - every client has a number; this is the list of all of them. The number of clients is **nclient**.

emp_client[nemp] - number of the client this employee is billed to

hoursworked[nemp][31] - for each employee, and for each day of the month, the number of hours worked

To get you started thinking about cells rather than loops, I am going to suggest that you use C-style pseudocode written in a way that is easily translatable into J. Your progress in J will be measured by how little you have to use this crutch.

Problem 1: How many hours did each employee work? The C code for this is:

```
void emphours(hrs)
int hrs[ ];  // result: hrs[i] is hours for employee i
{
    int i, j;
    for(i = 0;i<nemp;++i)
        for(j = 0,hrs[i] = 0;j<31;++j)hrs[i] += hoursworked[i][j];
}
```

The first step in translating this into J is to write the loops, but without loop indexes: instead, indicate what elements will be operated on:

```
for (each employee)
    for(each day)take the sum of hoursworked
```

Now, figure out what ranks the operands have. The **hoursworked** items that are added are scalars, so we will be looping over a list of them; that means we want the sum of items of a rank-1 list. So the inner loop is going to be `+/"1` . What about the outer loop? The information for each employee has rank 1 (each employee is represented in **hoursworked** by a single row), so a verb applied to each employee should have rank 1. Note that we don't worry about the actual shape of `hoursworked`—once we figure out that our verb is going to operate on 1-cells, we let J's implicit looping handle any additional axes. We build up the loops by applying the rank conjunction for each one, so we have the inner loop `+/"1` and the outer loop of rank 1; combined, they are `+/"1"1` . The `"1"1` can be changed to a single `"1`, and we get the final program:

```
emphours =: monad : '+/"1 hoursworked'
```

Problem 2: How much did each employee earn in wages? The C code is:

```
void empearnings(earns)
int earns[ ];  // result: earns[i] is wages for employee i
{
    int i, j;
    for(i = 0;i<nemp;++i) {
        for(j = 0,earns[i] = 0;j<31;++j)earns[i] += hoursworked[i][j];
        earns[i] *= payrate[i];
    }
}
```

When we write the pseudocode, we will change the algorithm just a bit: rather than multiplying each total by the billing rate just after the total is calculated, we will make one loop to calculate the totals, and then a second pass to multiply by the billing rate. This is a case where good J practice differs from good C practice. Because of the implicit looping that is performed on all verbs, you get better performance if you let each verb operate on as much data as possible. You may at first worry that you're using too much memory, or that you might misuse the processor's caches; get over it. Apply verbs to large operands. The pseudocode is:

```
for (each employee)
    for(each day)take the sum of hoursworked
for(each pair of wage_rate and sum)multiply the pair
```

The first two loops are just `+/"1 hoursworked` as before. The last loop clearly multiplies scalars, so it is `*"0` . We note that dyad `*` has rank 0, so we don't need to specify the rank, and we get the final program:

```
empearns =: monad : 'payrate * +/"1 hoursworked'
```

Problem 3: How much profit did each employee bring in? C code:

```
void empprofit(profit)
int profit[ ];  // result: profit[i] is profit from employee i
{
    int i, j, temp;
    for(i = 0;i<nemp;++i) {
        for(j = 0, temp = 0;j<31;++j)temp += hoursworked[i][j];
        profit[i] = temp * (billrate[i] - payrate[i]);
    }
}
```

Again, we create a new loop to calculate the list of profit for each employee:

```
for (each employee)
    for(each day)take the sum of hoursworked
for (each employee)take billing_rate - wage_rate;
for(each pair of profit and sum)multiply the pair
```

The profit is clearly a difference of scalars applied to two lists, therefore it will be `-"0` or equivalently simply `-` . The program then is

```
empprofit =: monad define
(billrate - payrate) * +/"1 hoursworked
)
```

---

7. More Verbs

Before we can write more complex programs, we need to learn some more verbs. We will group them into classes and give a few mnemonic hints.

Before we start, I should point out a convention of J: if a dyadic verb is asymmetric, you should think of `x` as operating on `y`, i. e. `x` is control information and `y` is data. We will note the exceptions to this rule—`%`, `/:`, `\:`, `-`, `-.`, and `e.`—and the reasons for the exception.

Arithmetic Dyads

All these verbs have rank 0 and produce a scalar result, so if they are applied to two operands of equal shape the result will also have that shape; if applied to two operands that agree, the result has the shape of the larger operand.

`x + y` addition

`x - y` subtraction (`y` operates on `x` to match the mathematical definition)

`x * y` multiplication

`x % y` division Note that the slash has another use, so `%` is division. You don't have to worry about division by zero: it produces _ (infinity) or __ (negative infinity) except for `0%0`, which yields `0` .

`x ^ y` exponentiation (`x` to the power `y`). `0^0` yields `1` .

`x ^. y` logarithm (base-`x` logarithm of `y`)

`x | y` modulus (remainder when `y` is divided by `x` . For the longest time I had trouble remembering dyad `|`; it seemed that the divisor should be `y` by analogy to dyad `%` . The inconsistency is that J defines `x % y` as '`x` divided by `y`' to match accepted practice in mathematics; that makes dyad `%` anomalous in J, because we have `y` operating on `x`).

The comparison verbs have rank 0, and produce Boolean results in which `1` means true, `0` means false. They use ***tolerant comparison***, which means that two values that are very close to equal are considered equal. This saves you the trouble of

adding small amounts to mask the effects of floating-point rounding:
`1 = 3 * 1 % 3` is **`1`**, unlike **`1.0 == 3.0 * 1.0 / 3.0`** whose value depends on the compiler. If you need exact comparison, append **`!.0`** to the verb. Look under 'Comparison Tolerance' for details.

`x = y` equal

`x ~: y` not equal (if you squint the colon looks like an equal sign)

`x > y` greater-than

`x < y` less-than

`x >: y` greater-than or equal (if you squint the colon looks like an equal sign)

`x <: y` less-than or equal

Boolean Dyads

These verbs have rank 0 and are applied to Boolean arguments to produce Boolean results:

`x *. y` Boolean AND

`x +. y` Boolean OR

`x = y` Boolean XNOR (1 if the operands are equal)

`x ~: y` Boolean XOR (1 if operands differ)

Min and Max Dyads

These verbs are useful for performing tests, because they perform the operation item-by-item, replacing a C loop that does a test for each atom. They have rank 0 and produce a scalar in each cell, so the result has the shape of the larger operand.

`x >. y` the greater of **`x`** and **`y`**

`x <. y` the lesser of **`x`** and **`y`**

Arithmetic Monads

These verbs have rank 0.

`>: y` increment (**`y+1`**)

`<: y` decrement (**`y-1`**)

`<. y` the largest integer not greater than y (floor function)

`>. y` the smallest integer not less than y (ceiling function)

`| y` absolute value of **`y`**

`* y` signum of **`y`** (_**`1`** if **`y`** is negative, **`0`** if **`y`** is tolerantly close to 0, **`1`** if **`y`** is positive)

`x o. y` trigonometric function. Think of dyad **`o.`** as a monad, selecting the function based on **`x`** . For example, **`1 o. y`** is *sin*(*y*), and _**`3 o. y`** is *arctan* (*y*). The functions are: **`0`** *sqrt*(1-*sqrt*(*y*)); **`1`** *sin*(*y*); **`2`** *cos*(*y*); **`3`** *tan*(*y*); **`4`** *sqrt* ((*y***y*)+1); **`5`** *sinh*(*y*); **`6`** *cosh*(*y*); **`7`** *tanh*(*y*); **`8`** *sqrt*(-(1+*y***y*)); **`9`** Re(*y*); **`10`** Mag(*y*); **`11`** Im(*y*); **`12`** Angle(*y*). **`(-x) o`** is the inverse of **`x o`**, with the exceptions _**`8`** -*sqrt*(-(1+*y***y*)); _**`9`** *y*; _**`10`** Conj(*y*); _**`11`** **`j. y`**; _**`12`** **`^j. y`** . For mnemonic purposes, note that the odd numbers specify odd functions.

`-. y 1-y`

Boolean Monad

`-. y` (rank 0) Negate **`y`** This is simply the boolean interpretation of **`1-y`** .

Operations on Arrays

These verbs operate on entire arrays; they have infinite right rank (so they look at the entire **`y`**), and they have left rank as appropriate for the operation performed. I am going to give a highly simplified definition of the functions of these verbs; consult the Dictionary to see all they can do.

Dyads

Selection: `{ # -.`

`x { y` (From)

No, nothing was left out! **`{`** is not paired with **`}`**; it is a verb and stands by itself.

The left rank is 0; the result is item number **`x`** of **`y`** . One of the great insights of J is the realization that selection, which in most other languages requires a special syntax like C's **y[x]**, is really just a dyadic verb like any other. Examples:

```
   2 { 3 1 4 1 5 9
4
```

`y` has rank 1; its items have rank 0; item number 2 is `4` .

```
   2 4 { 3 1 4 1 5 9
4 5
```

The left frame is `2`; each atom of `x` selects an item of `y` (each an atom), and the results are made into a 2-item list.

```
   1 { i. 3 3
3 4 5
```

`y` has rank 2; its items have rank 1; item number 1 is `3 4 5` .

```
   2 1 { i. 3 3
6 7 8
3 4 5
```

The left frame is `2`; each atom of `x` selects an item of `y` (each a list), and the results are made into a 2×3-item array.

```
 0 { 5
5
```

You are allowed to select item 0 of a scalar. This follows from the definition that a scalar has a single item which is the scalar itself. Selecting any other item is an error.

There are many variations on the format of `x`, providing for multidimensional indexing where each index can be a list. You will have to wait a bit to learn them, but I will note here that a negative index counts back from the end of the array:

```
   _1 { 3 1 4 1 5 9
9
```

`x # y` (Copy)

Left rank is 1. If `x` is a list whose items are all 0 or 1, the result has the same rank as `y`, and contains just the items of `y` for which the corresponding item of `x` is 1. For example:

```
 1 0 1 0 0 0 # 3 1 4 1 5 9
3 4
   0 0 1 # i. 3 4
8 9 10 11
```

`x -. y` (Remove)

Left rank is infinite. Any items of `x` that match items of `y` are removed:

```
   1 2 3 4 5 4 3 -. 2 4
1 3 5 3
```

The verb is defined with `y` operating on `x` because of the analogy with `-` . If `x` and `y` are sets, `x -. y` is the set difference.

Indexing: `i. e.`

`x i. y` (Index Of) and `x i.!.0 y` (Intolerant Index Of)

The left rank is infinite. `x i. y` looks through the items of `x` to find one that matches `y`; the result is the item number of the first match. Examples:

```
   3 1 4 1 5 9 i. 5
4
   (i. 4 3) i. 6 7 8
2
```

It may occur to you that for `y` to match an item of `x`, the rank of `y` must be the same as the rank of an item of `x` (call that ***rix***, which is one less than the rank of `x` (namely `#$x`) unless `x` is an atom; formally, *rix* is `(0 >. (#$x) - 1)` or, more cleverly, `#${.x`). If `y` is of higher rank, each *rix*-cell of `y` is matched against items of `x` . Formally, `x i. y` is equivalent to `x i."(_,rix) y` . Example:

```
   3 1 4 1 5 9 i. 1 5
1 4
   (i. 3 3) i. (i. 2 3)
0 1
```

If an *rix*-cell of `y` matches none of the items of `x`, the result value for that cell is `#x`, i. e. one more than the largest valid item-number of `x` :

```
   3 1 4 1 5 9 i. 8 4 _1
6 2 6
```

To be hair-splittingly accurate we must say that `x i. y` is equivalent to `x i."(_,rix)"_ y` because the rank of dyad `i.` is infinite. This distinction will matter eventually.

The comparison is dyad `i.` is tolerant, that is, numeric operands that are very close to equal are considered equal. The special form `i.!.0` is like `i.`, except that the comparison is intolerant. `i.!.0` uses a different algorithm from `i.`, and may be faster even if the operands are not numeric.

`x e. y` (Element Of)

The left rank is infinite. `x e. y` is a lightweight version of `y i. x`; the result is 1 if `x` matches an item of `y`, 0 if not. The verb is applied to *riy*-cells of `x` (where ***riy*** is the rank of an item of `y`). Formally, `x e. y` is the same as `(#y) ~: y i. x` . Dyad `e.` is an exception to the rule that `x` is control information and `y` is data. It was defined to be reminiscent of mathematical epsilon meaning 'element of'.

Take and Drop: `{. }.`

`x {. y` (Take)
`x }. y` (Drop)

The left rank is 1, but the verb handles scalars also; we will consider only the case where `x` is a scalar. `x {. y` (***take***) takes the first `x` items of `y`, i. e. it produces a result which consists of the first `x` items of `y`; `x }. y` (***drop***) discards the first `x` items of `y` . If `x` is negative, `x {. y` takes the last (`|x`) items of `y`, and `x }. y` discards the last (`|x`) items (remember, `|x` is the absolute value of `x`). The rank of the result is always the same as the rank of `y`, and in all cases the order of items is unchanged. Examples:

```
   2 {. 3 1 4 1 5 9
3 1
   2 }. 3 1 4 1 5 9
4 1 5 9
   _2 {. 3 1 4 1 5 9
5 9
   _1 }. i. 3 3
0 1 2
3 4 5
```

`x {. y` always gives you as many items as you asked for. If you ***overtake*** by asking for more than `#y` items, J will create extra ones, filling them with `0` or `' '` as appropriate:

```
   5 {. 3 1 4
3 1 4 0 0
   _5 {. 'abc'
  abc  NB. Negative overtake: fills added at front
```

We have met fills before; they were added to bring the results from different cells of a verb up to a common shape so that they could be made into an array. The fills added by overtake are different: they are part of the execution of the verb itself. We will distinguish the two types of fill, calling the ones added by the verb itself ***verb fills*** and the ones added to make cell-results compatible ***framing fills***. Framing fills are always `0` or `' '`, but you can specify the value to use for a verb fill, using the ***fit conjunction*** `!.` :

```
   5 {.!.9 (3 1 4)
3 1 4 9 9
   _5 {.!.'x' 'abc'
xxabc
```

The fit conjunction creates a new verb; in this case `{.!.f` is a verb that looks just like `{.` but uses `f` for the verb fill.

The fit conjunction is by no means reserved for specifying verb fills: it is available for use on any primitive to make a small change to the operation of the primitive. If `!.` has a meaning for a primitive, that meaning is given in the Dictionary entry for the primitive.

Joining Arrays: `,.` `,:`

`x ,. y` (Stitch)

The left rank is infinite. `x ,. y` is equivalent to `x ,"_1 y` . That means that dyad `,` is applied to the corresponding items of `x` and `y`, making each item of the overall result the concatenation of the corresponding items. Example:

```
   3 4 5 ,. 7 8 9
3 7
4 8
5 9
```

`x ,: y` (Laminate)

The left rank is infinite. `x ,: y` is a list of 2 items: item 0 is `x` and item 1 is `y` .

If `x` and `y` do not have the same shape, they are brought to a common rank and padded with fills to a common shape. Dyad `,:` concatenates `x` and `y` along an added axis, in contrast to dyad `,` which concatenates them along their leading axis:

```
   3 4 5 ,: 7 8 9
3 4 5
7 8 9
```

Contrast this with dyad `,.` above or dyad `,` which would produce

`3 4 5 7 8 9` .

```
   1 2 , 3 4 ,: 5 6
1 2
3 4
5 6
```

Take a moment to understand why in this example the first verb is dyad `,` and the second is dyad `,:` .

Rotate Left and Shift Left: |.

`x |. y` (Rotate Left)

The left rank is 1, but we will discuss only the case where `x` is a scalar. The result has the same shape as `y`, with the items of `y` rotated `x` places to the left (with wraparound). If `x` is negative, the items are rotated to the right. Examples:

```
   2 |. 3 1 4 1 5 9
4 1 5 9 3 1
   _1 |. i. 3 3
6 7 8
0 1 2
3 4 5
```

`x |.!.f y` (Shift Left)

When the fit conjunction is used, any item that is rotated across the beginning of `y` is replaced by the fill `f` . This turns the rotate into a shift where `f` gives the value to be shifted in:

```
   2 |.!.'x' 'abcde'
cdexx
   _2 |.!.'x' 'abcde'
xxabc
```

Sort: `/:` `\:`

`x /: y` (Sort Up Using)
`x \: y` (Sort Down Using)

The left rank is infinite. **`x`** and **`y`** must have the same number of items. The items of **`x`** are *records* and the items of **`y`** are *keys*; **`x /: y`** is the records **`x`** sorted into ascending order of corresponding keys **`y`** . **`x \: y`** sorts into descending order. Examples:

```
   3 1 4 1 5 9 /: 0 1 2 3 4 5
3 1 4 1 5 9
```

`y` was already in order.

```
   3 1 4 1 5 9 /: 5 4 3 2 1 0
9 5 1 4 1 3
```

Sorting into reverse order.

```
   3 1 4 1 5 9 /: 0 10 1 20 2 30
3 4 5 1 1 9
```

`x` in order of ascending **`y`** .

```
   (i. 4 3) /: 10 20 1 2
6  7  8
9 10 11
0  1  2
3  4  5
```

Items of **`x`** in order of ascending **`y`** .

```
   1 3 5 /: 7 8 , 1 2 ,: 4 5
3 5 1
```

The keys **`y`** do not have to be single numbers; they don't even have to be numeric. Here, **`1 2`** is lowest, then **`4 5`**, then **`7 8`** . The Dictionary gives complete rules for ordering **`y`** .

Because sorting is not so easy in C, C programmers are not quick to recognize applications of **`/:`** and **`\:`** . The J implementation of **`/:`** and **`\:`** runs in linear time for most **`y`** and should not be avoided.

According to our general principle, we would expect that in dyad **`/:`** **`x`** held the keys and **`y`** the data. Dyad **`/:`** is an exception to the rule.

```
y /: y
y \: y
```

When `x` and `y` are the same, you have the simple case of sorting `y` into ascending or descending order.

Match: `-:`

`x -: y` (Match)

The left rank is infinite. The result is `1` if `x` and `y` are the same, `0` otherwise, **except**: (1) if they are numeric, the comparison is tolerant; (2) for some reason I don't understand, if they are empty, they are considered to match even if the types are different, which means that getting a result of `1` from `x -: y` is no guarantee that `x` and `y` will behave identically:

```
   (0$0) -: ''
1
   (1 {. 0$0) -: (1 {. '')
0
```

The important difference between dyad `-:` and dyad `=` is that dyad `-:` has infinite rank so you get a single result covering the entire array, and it won't fail if the shapes of `x` and `y` do not agree.

Monads

Enfile: `,`

`, y` (Enfile)

`, y` consists of all the atoms of `y`, made into a list. The order is row-major order , i. e. all the atoms of item 0 of the original `y` come first, followed by atoms of item 1, and so on; within each item the ordering similarly preserves the order of subitems. Examples:

```
   , i. 2 3
0 1 2 3 4 5
```

The atoms were made into a list.

```
   a =. 2 2 3 $ 'abcdefghijkl'
   a
abc
def
```

```
ghi
jkl
   ,a
abcdefghijkl
```

`y` of any shape produces a list.

Recall that a single quoted character is an atom rather than a list:

```
   $'x'

```

To get a 1-character list, use monad `,` :

```
   $,'x'
1
```

The official name for monad `,` is the quaint but unedifying 'ravel', meaning 'separate or undo the texture of'. I prefer the equally quaint but more descriptive 'enfile', which means 'arrange in a line (as if on a string)'.

Reverse and Transpose: `|.` `|:`

`|.` `y` (Reverse)

The items of `y` are put into reverse order:

```
   |. i. 5
4 3 2 1 0
```

`|:` `y` (Transpose)

The axes of `y` are reversed. This is difficult to visualize for high ranks but easy for the most common case, rank 2:

```
   |: i. 3 4
0 4  8
1 5  9
2 6 10
3 7 11
```

Take and Drop Single Item: `{.` `{:` `}.` `}:`

`{.` `y` (Head)

`{:` `y` (Tail)

`}.` `y` (Behead)

`}:` `y` (Curtail)

The operations performed are simple; the biggest problem is remembering which primitive does what. Remember that **`{`** means *take* and **`}`** means *drop*, and that **`.`** means *beginning* and **`:`** means *end*. So, **`{.y`** is the first item of **`y`**, **`{:y`** is the last item of **`y`**, **`}.y`** is all of **`y`** except the first item, **`}:y`** is all of **`y`** except the last item:

```
   {. 3 4 5
3
   {: i. 3 4
8 9 10 11
   }. 3 4 5
4 5
```

`}.y` is identical to **`1}.y`** and **`}:y`** is identical to **`_1}.y`** . **`{.y`** is **not** identical to **`1{.y`**, because **`1{.y`** has the same rank as **`y`** while **`{.y`** has the rank of an item of **`y`** .

Grade (Create Ordering Permutation): `/:` `\:`

`/: y` (Grade Up)
`\: y` (Grade Down)

`/: y` creates a numeric list with **`#y`** items, such that **`i{/:y`** is the index of the **`i`**th-largest item of **`y`** . **`(/:y){y`** gives **`y`** sorted into ascending order. **`/:y`** is a ***permutation vector***, i. e. it contains each integer in the range **`0`** to **`(#y)-1`**. **`\:`** is similar but works in descending order. Example:

```
   /: 3 1 4 1 5 9
1 3 0 2 4 5
```

Read this result as follows: the smallest item of **`y`** is item 1 (with value 1), the next-smallest is item 3 (1), then item 0 (3), then item 2 (4), then item 4 (5), then item 5 (9). Monad **`/:`** defined this way turns out to be surprisingly useful. As a limbering-up exercise in the use of permutation vectors, and an example of how compactly J can express ideas, see if you can describe in words what *two* applications of monad **`/:`** will give:

```
   /: /: 3 1 4 1 5 9
2 0 3 1 4 5
```

Add An Axis: `,:` (Itemize)

`,: y`

The result of `,: y` has rank one higher than the rank of **`y`**, with one item, which is **`y`** . The shape of `,: y` is the shape of **`y`** with 1 prepended (in plain J, **`$,:y`** is **`1,$y`**).

Constant Verb

m"n

We have met the rank conjunction **`"`** applied to verb left arguments; applied to noun left arguments it produces a verb whose result is always the value of the noun. The created verb (which can be used as either a monad or a dyad) has ranks, given as the right operand of **`"`** . Examples:

```
   5"_ i. 4 4
5
```

The simplest and most common case. Since the verb has infinite rank, it operates on the entire operand and produces the scalar value.

```
   5"0 i. 3
5 5 5
```

Here the verb is applied to each 0-cell of the list, giving the scalar result for each cell.

```
   1 2 3"0 i. 3
1 2 3
1 2 3
1 2 3
```

Here the result at each cell is the left argument of **`"`** , the list **`1 2 3`** . If you were expecting the **`3`** to be repeated, remember that you can look at the words as J sees them:

```
   ;: '1 2 3"0 i. 3'
+-----+-+-+--+-+
|1 2 3|"|0|i.|3|
+-----+-+-+--+-+
```

`1 2 3` is a single word.

_9:…_1: 0: 1:…9: _: __:

For a few special values, namely the integers **`_9`** through **`9`**, infinity _, and negative infinity __, you can create an infinite-rank verb to produce the value by following the constant value with **`:`** . This is equivalent to ***`value`*`"_`** .

```
   3: 'abc'
3
```

The operand **`'abc'`** was ignored, and the result was **`3`** .

8. Loopless Code II—Adverbs / and ~

The monad **`+/`**, which sums the items of its operand, is a special case of the use of the adverb **`/`** . It is time to learn about adverbs, and other uses of this one.

Modifiers

An adverb is a modifier. It appears to the right a noun or verb; the prototype is **`u a`** where **`u`** is the noun or verb and **`a`** is the adverb. The ***compound*** **`u a`** is a new entity, and not necessarily the same part of speech as **`u`** . When the compound **`u a`** is executed, it performs the function given by the definition of **`a`** and has access to **`u`** during its execution. If **`u a`** is a verb, then it also has access to the operands of the verb during its execution; the verb **`u a`** will then be invoked as **`u a y`** if monadic or **`x u a y`** if dyadic.

You will note that I didn't have to write **`x (u a) y`** . While J gives all verbs equal precedence and executes them right-to-left, it does give modifiers (adverbs and conjunctions) higher precedence than verbs, in the same way that C and standard mathematical notation give multiplication precedence over addition. We will discuss the parsing rules in detail later; for now, know that modifiers are bound to their operands before verbs are executed, and that if the left operand of a modifier has a conjunction to its left (e. g. **`x c y a`**), the conjunction is bound to its arguments first, and the result of that becomes the left argument to the modifier: **`x c y a`** is **`(x c y) a`**, not **`x c (y a)`** . In other words, modifiers associate left-to-right. So, **`+"1/`** (in which **`"`** is a conjunction and **`/`** is an adverb) is the same as **`(+"1)/`**, **not** **`+"(1/)`** . The phrase **`| +/"1 (4) + i. 3 3`** is executed as **`| ((+/"1) ((4) + (i. 3 3)))`**, in accordance with the rule: right-to-left among verbs, but applying modifiers first. Note that I had to put parentheses around the **`4`**, because **`"1 4`** would have been interpreted as rank **`1 4`** : collecting adjacent numbers into a list is done before anything is executed.

J includes a rich set of modifiers and even allows you to write your own, though many J programmers will never write a modifier. We will begin our study of modifiers with the adverb monad **`u/`** which goes by the mnemonic 'Insert'.

The Adverb Monad `u/`

Monad **`u/`** (by which we mean **`/`** with a verb left operand, used as **`u/ y`** rather than as **`x u/ y`** which is dyad **`u/`** and is completely different; note that **`m/ y`** where **`m`** is a noun is different yet), inserts **`u`** between items of **`y`** . Monad **`u/`** has infinite rank. As a simple example, **`+/ 1 2 3`** is equivalent to **`1 + 2 + 3`** :

```
   +/ 1 2 3
6
```

As usual, we can use **`fndisplay`** to explain what's happening:

```
   defverbs 'plus"0'
   plus/ 1 2 3
+---------------+
|1 plus 2 plus 3|
+---------------+
```

The great power of the adverb concept is that **`u`** can be any verb; it's not restricted to **`+`**, **`-`**, or any other subset of verbs (it can even be a user-written verb). What would monad **`>./`** mean? Well, **`>./ 1 2 3`** would be equivalent to **`1 >. 2 >. 3`**; since each **`>.`** picks the larger operand, the result is going to be the largest number; so monad **`>./`** means 'maximum':

```
   >./ 3 1 4 1 5 9
9
```

and of course 'minimum' is similar:

```
   <./ 3 1 4 1 5 9
1
```

What about monad **`,/`**? Convince yourself that it combines the first two axes of its operand:

```
   ,/ i. 2 3
0 1 2 3 4 5
   defverbs 'comma'
   comma/ i. 2 3
+-------------------+
|(0 1 2) comma 3 4 5|
+-------------------+
   i. 2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
   $ ,/ i. 2 3 4
6 4
   ,/ i. 2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
```

How many atoms are in `y`? Why, `*/ $ y` :

```
   */ $ i. 2 3 4
24
```

We can verify that the rows and columns of the following magic square sum to the same value:

```
   +/ 3 3 $ 8 1 6  3 5 7  4 9 2
15 15 15
   +/"1 (3 3) $ 8 1 6  3 5 7  4 9 2
15 15 15
```

As this last example shows, the items can be of any shape. Applying `+/` to the rank-2 array added up 1-cells, while applying `+/"1` added up the 0-cells within each 1-cell.

Have you wondered what would happen if there is no cell or only 1? Good on you if you did. The answer is: if there is only 1 cell, the result is just that cell; if there is no cell, the result is a cell of identity elements. The ***identity element*** `i` for a dyadic verb `v` is that value of `i` such that `i v y` is `y` for any choice of `y` . For example, the identity element for `+` is `0`, because `0 + y` is always `y` . The identity element for `*` is `1`, and for `<.` is _ . If there is no identity element for a verb `v` (for example, `$` has no identity element), you will get a domain error if you apply `v/` to an empty list. Examples:

```
   +/ 0$0
0
   */ 0$0
1
```

Empty list; result is the identity element.

```
   +/ 1 3 $ 3 5 7
3 5 7
```

There is 1 1-cell, so the result is that cell. This result has shape `3`, **not** `1 3` .

```
   +/ 0 3 $ 0
0 0 0
```

There are 0 1-cells, so the result is a cell of identity elements. Note that even though there are no cells, the cell still has a shape which is made visible by `+/` .

```
   $/ 0$0
|domain error
|       $/0$0
```

If you don't want to figure out what an identity element for a verb `v` is you can ask the interpreter by typing `v/ 0$0` .

Before we move on you should note that since `v/ 1 2 3` is equivalent to `1 v 2 v 3`, `2 v 3` is evaluated first: the operation starts at the end of the list and moves toward the beginning.

The adverb ~

`~` is an adverb. Like all adverbs, it has a monadic and dyadic form. The dyadic form `x u~ y` is equivalent to `y u x`; in other words, the operands of `u` are reversed. The ranks of dyad `u~` are the same as those of dyad `u`, but with left and right rank interchanged. For advanced J dyad `~` is indispensable; even in ordinary use it can save time and obviate the need for parentheses:

```
   (10 + 2) % 3
4
   3 %~ 10 + 2
4
```

Using `%~` to mean '`y` divided by `x`', we can have right-to-left execution without parentheses.

```
   -~/ 2 4
```

```
2
```

When we know `y` contains exactly 2 items, `-/ y` is a convenient shorthand to subtract the second from the first without having to write `({.y) - ({:y)` . To subtract the first from the second, we simply invert the order of subtraction with `-~` .

The monadic form `u~ y` has infinite rank and is equivalent to `y u y`, i. e. it applies *dyad* `u` with both the left and the right operands equal to the one operand of monad `u~` . As with dyad `u~`, most uses of monad `u~` are esoteric, but we know one already: we can sort `y` into ascending order with either `y /: y` or our new equivalent `/:~ y` :

```
   /:~ 3 1 4 1 5 9
1 1 3 4 5 9
```

---

9. Continuing to Write in J

Now that we have a formidable battery of verbs at our command, let's continue writing programs in J. The data definitions are repeated here for convenience:

empno[nemp] (in J, just **empno**) - employee number for each member of the staff. The number of employees is **nemp**.

payrate[nemp] - number of dollars per hour the employee is paid

billrate[nemp] - number of dollars per hour the customer is billed for the services of this employee

clientlist[nclients] - every client has a number; this is the list of all of them. The number of clients is **nclient**.

emp_client[nemp] - number of the client this employee is billed to

hoursworked[nemp][31] - for each employee, and for each day of the month, the number of hours worked

Problem 4: Find the amount to bill a given client. C code:

```
int billclient(cno)
int cno;  // number of client we are looking up
{
    int i, j, temp, total;
    total = 0;
    for(i = 0;i<nemp;++i) {
        if(emp_client[i]==cno) {
            for(j = 0, temp = 0;j<31;++j)temp += hoursworked[i][j];
        total += billrate[i] * temp;
    }
return(total);
}
```

The function is implemented in C by looping over the employees and picking the ones that are working for the specified client. In J we deal with entire arrays rather than with elements one at a time, and the general plan is:

```
get the mask of employees billed to the requested client;
select the hoursworked records for the applicable employees;
for(each employee) // 1
    for(each day) accumulate total hours; // 2
for(each employee)multiply hours by billing rate;
for(each employee)get total billing;
```

This example is the first one in which an argument is passed into the J verb. Within the verb the right argument is referred to as `y.` and the left argument (if the verb is a dyad) is referred to as `x.` . As usual in C we might use fewer big loops, but in J we stick to small loops. The mask of employees billed to the client is given by `emp_client = y.` which is a mask with `1` for the selected employees, `0` for the others (remember that `=` is a test for equality, not an assignment). We can select the `hoursworked` items for the specified client by `(emp_client = y.) # hoursworked`; then the sum for each day will be a sum within each 1-cell, resulting in a list of hours for each selected employee. The line `+/"1 (emp_client = y.) # hoursworked` performs the functions of loops 1 and 2 in the pseudocode: loop 1 within each cell and loop 2 in the implied loop over the cells. Then, it is a simple matter to select the billing rates for the employees, multiply each billing rate by the number of hours worked, and take the total over all employees billed to the customer. The solution (using a temporary variable to hold the mask) is

```
billclient =: monad define
mask =. emp_client = y.
+/ (mask # billrate) * +/"1 mask # hoursworked
)
```

Problem 5: For each day, find the worker who billed the most hours. C code:

```
int dailydrudge(drudges)
int drudges[31]; // result: empno of worker with most hours each day
{
    int i, j, highhours;
    for(i = 0;i<31;++i) {
        highhours = -1;
        for(j = 0;j<nemp;++j)
            if(hoursworked[j][i]>highhours) {
                drudges[i] = empno[j];
                highhours = hoursworked[j][i];
            }
         }
      }
   }
```

We note that the inner loop, which records the employee number of any employee who worked more than the previous high, does not correspond to any of our J verbs. So, we break this loop into operations that do correspond to J verbs:

```
for(each day)find the maximum number of hours worked; // 1
for(each day)find the index of the employee who worked that much; // 2
for(each day)translate the index to an employee number; // 3
```

Loop 1 is simply `>./ hoursworked` . Loop 2 calls for dyad `i.` to perform the search, but there is a little problem: each search must go through a column of `hoursworked`, giving the hours worked by each employee on that day, but the column is not an item of `hoursworked`; each item is a row, giving hours worked on each day by one employee. The solution is to transpose `hoursworked` (by `|: hoursworked`), making the items correspond to days rather than employees. Then we match up the resulting1-cells with the 0-cells of the maximum found by loop 1 and find the index of each maximum, using `i."1 0` or the equivalent `i."_1` . Loop 3 is a simple application of dyad `{` . The final code is

```
dailydrudge =: monad define
((|: hoursworked) i."_1 >./ hoursworked) { empno
)
```

Problem 6: Order the employees by the amount of profit produced. This is a bagatelle for J and we won't even bother with C code. We have a verb that returns the profit for each employee, so we call it and use the result as the keys for sorting the employee numbers into descending order. Note that a verb must be given an argument when it is executed; we use `0` as a convenient value. The final J code is

```
producers=: monad : 'empno \: empprofit 0'
```

which makes use of the verb we defined earlier:

```
empprofit =: monad define
(billrate - payrate) * +/"1 hoursworked
)
```

Problem 7 is similar: Order the clients by the amount of profit produced. It requires more ingenuity, and the C code would be more than I want to show, so let's try to write in J directly. We start with the list of clients `clientlist`, the list that tells which client each employee worked for `emp_client`, and the profit per employee given by `empprofit 0` . For each client, we need to find the employees that worked for the client and add up the profit they brought in. For this kind of problem we want an `array` with employees for one axis and clients for the other, with a 1 in the positions where the employee is assigned; then we can do `array * empprofit 0` and do some suitable summing of the result. Let's work out what `array` must look like. Since dyad `*` has rank 0, `array * empprofit 0` is going to replicate 0-cells of the shorter-frame argument (`empprofit 0`) which means that a single profit value is going to multiply an entire 1-cell of `array` . So we see that the leading axis of `array` must be clients and the second axis must be employees; each item will have the client mask for one employee. The way to create that is by `clientlist ="1 0 emp_client` which will compare the entire client list against each 0-cell of `emp_client` and form the results into an array. Then, `(clientlist ="1 0 emp_client) * empprofit 0` will have one item per employee, each item having the amount billed to each client; we sum these items to produce a list with the profit from each client, and use that to order the client numbers. Solution:

```
custbyprofit =: monad define
clientlist \: +/ (clientlist="1 0 emp_client) *
empprofit 0
)
```

For our final problem in the payroll department, consider how to calculate withholding tax on each employee's earnings. The tax rate within tax brackets will be assumed to be the same for all employees. C code for this would look like:

```
int renderuntocaesar(shekels)
float shekels[ ]; // result: withholding for each employee
{
    // tax-bracket table: start of each bracket, ending with high value
    float bktmin[4] = {0,10000,20000,1e9};
    // tax-bracket table: rate in each bracket
    float bktrate[3] = {0.05,0.10,0.20};
    int earns[nemp];
    int i, j;
    empearnings(earns);  // get earnings for each employee
    for(i = 0;i<nemp;++i) {
        shekels[i]= 0.0;
        for(j = 0;j<sizeof(bktrate)/sizeof(bktrate[0]);++j) {
            if(earns[i] > bktmin[j]) {
                float bktval = bktmin[j+1];
                if(earns[i] < bktval)bktval = earns[i];
                shekels[i] += bktval * bktrate[j];
            }
        }
    }
}
```

In J, we will sum over the tax brackets and we must create a suitable array for the summation. The items in this array will be the amounts earned in each tax bracket. Corresponding to the two `if` statements we will use conditional operators to discard amounts earned outside the tax bracket. The conditionals will operate on each row, so they will have rank 1, and they will look something like `0 >. (bracket_top <. earned) - bracket_bottom` which will give the amount earned in the bracket, set to 0 if the earnings are below the bracket and to the size of the bracket if above. We will create `bracket_top` by shifting the brackets left and filling with infinity (this corresponds to the `bktmin[j+1]` reference in the C code). We could create `earned` by replicating the earnings for each employee to form a 1-cell for each bracket—the code for this would be `(#bktmin) $"0 empearnings ''`—but it's not necessary to create that array explicitly: we just use the implicit looping to cause each cell of `empearnings ''` to be replicated during the comparison with `bracket_top`. Making all these substitutions, noting that all the operations have rank 1, and summing at the end over the items within each 1-cell, we get the final code:

```
renderuntocaesar =: monad define
bktmin =. 0 10000 20000
bktrate =. 0.05 0.10 0.20
t=. ((1 |.!._ bktmin) <."1 0 empearnings '') -"1 bktmin
+/"1 bktrate *"1 (0) >. t
)
```

We used the temporary variable `t` so our sentences would fit on the page.

Let's write a program to count the lines and words in a file. This is a simple task, and in C it might look like:

```
int[2] wc(f)
```

```
char *f;  /* pointer to filename */
{
     FILE fid;
     int ct[2];  /* # words, # lines */
     char c;

     fid = fopen(f);
     while(EOF != (c = fgetc(fid)) {
          if(c == ' ')++ct[0];
          if(c == LF)++ct[1];
     }
     return (ct);
}
```

Rather than loop for each character, in J we process files by reading the whole file into a list of characters and then operating on the list. The monad **`ReadFile`** (defined in the script **`jforc.ijs`**) has infinite rank; it takes a filename **`y`** and yields as result the contents of the file, as a list of characters. Once the file is a list, it is trivial to compare each character against space and linefeed, yielding for each a list of **`1`**s at each position filled by the character; then summing the list gives the number of each character. J code to do this is:

```
NB. y. is string filename, result is (#words),(#lines)
wc =: monad define
+/"1 (' ',LF) ="0 1 ReadFile y.
)
```

Suppose our user complains that **`wc`** needs improvement: specifically, it should also return the number of characters in the file, should not count multiple consecutive whitespace characters (including TAB) as separate words, and should treat the trailing LF as a word delimiter as well as a line delimiter. In C, we would respond by adding a few more tests and flags, but in J we realize that a major change to the program's function calls for a thorough rethinking.

To ignore multiple whitespace characters we need to define what an ignored whitespace is, without using flags and loops. This part of the problem often calls for creative thought; here we realize that a whitespace character is to be ignored if the previous character is whitespace. That's easy, then: we just calculate a Boolean vector, **`1`** if the character is whitespace, and then use shift and Boolean AND to ignore multiple whitespace. The code to do this is worth looking at; it is

```
sw =. (-. _1 |. w) *. w =. f e. ' ',TAB,LF
```

where `f` is the contents of the file. Note that we assign a value to `w` just before we right-shift `w` . This is legal in J: sentences are processed right-to-left, and the interpreter has not seen the reference to `w` at the time `w` is assigned. A similar statement in C, for example **x = w + (w = 4);** , would be undefined. Of course, even though it's legal in J, some would cavil—we will eventually learn ways to do this without defining a variable at all—but I leave it to you to decide how far you will detour to avoid jaywalking. Once `sw` has been calculated, the rest of the program is trivial. The final result is:

```
NB.Version 2.  Discard multiple whitespace,
NB.and return (#chars),(#words),(#lines)
wc2 =: monad define
f =. ReadFile y.
sw =. (-. _1 |. w) *. w =. f e. ' ',TAB,LF
(#f),(+/sw),(+/ LF = f)
)
```

10. Compound Verbs

On New Year's Day my rich uncle gives me **`x`** dollars, which I add to the **`y`** dollars I already have earning 4% interest. How much money will I have at the end of the year? Simple in J—I just write **`1.04 * x + y`**, and I have the answer, whether **`x`** and **`y`** are scalars or arrays. That's nice, but here's my problem: I expect his largesse to continue, and in my anticipation I have estimated his gifts for the next few years as the list **`x`**; I want to know what I'll be left with at the end. I need to pass each year's starting balance into the calculation for the next year. I know what I want the result to look like: it'll be **`v/ (|.x) , y`** which will be evaluated as **`xn v … x2 v x1 v x0 v y`** . But what is **`v`**? The problem with **`1.04 * x + y`** is that it contains 2 verbs and a constant, and I need it all lumped into a single verb so that I can have the adverb dyad **`/`** modify the whole thing. One solution would be to create the verb

```
v =: dyad : '1.04 * x. + y.'
```

after which **`v/ (|.x),y`** works, but it's a shame to have to interrupt a J sentence just to define a verb with such a puny function—I want magic words to let me say **`(1.04 * + abracadabra…combine!)/ (|.x),y`** . J has such magic words, and we will learn a few now.

The magic words will join verbs and nouns together, so they must be modifiers: adverbs and conjunctions. Before we start, we need a little notation to help with the different cases we will encounter. Given a conjunction **`c`** or adverb **`a`**, we call its left operand **`m`** if it is a noun, or **`u`** if it is a verb. Similarly we call a conjunction's right operand **`n`** if it is noun, **`v`** if a verb. There are four possible ways to invoke a conjunction (**`u c v`**, **`m c v`**, **`u c n`**, and **`m c n`**) and two for an adverb (**`u a`** and **`m a`**) and **they are defined independently**. Moreover, the ***derived verb*** produced by the invocation (the derived entity may be a noun, adverb, or conjunction too but that is unusual) can be used as a dyad (e. g. **`x u c n y`**) or as a monad (e. g. **`m a y`**), and those cases are defined independently as well. You won't get the cases mixed up, because verbs and nouns are so different that it will seem natural for **`u c n`** to be different from **`u c v`**; just be aware that the variants are many and that we will be learning a tiny subset of J's toolkit. The adverb **`/`** is an example: we have learned about monad **`u/`**, but dyad **`u/`** is very different, as is **`m/`** .

Verb Sequences—`u@:v` and `u@v`

`u@:v` creates a derived verb of infinite rank that applies `v` to its argument(s) and then applies `u` to the result. In other words, `u@:v y` is the same as `u v y` and `x u@:v y` is the same as `u x v y` . Examples:

```
   {. @: /: 3 1 4 1 5 9
1
```

Monad `/:` produced the permutation `1 3 0 2 4 5` of which we took the first item.

```
   1 2 3 +/@:* 1 2 3
14
```

Dyad `*` produced `1 4 9` whose items we then summed. `fndisplay` shows the details:

```
   defverbs 'plus"0 times"0'
   1 2 3 plus/@:times 1 2 3
+-------------------------------------------+
|(1 times 1) plus (2 times 2) plus 3 times 3|
+-------------------------------------------+
```

`u@v` is like `u@:v` except that the rank of the derived verb is the rank of `v` (also expressible as `(u@:v)"v` because `u"v` is defined to have the function of `u` with the rank of `v`). My advice is to stick to `@:` and avoid `@` unless you're sure you need it.

The Difference Between `u@:v` and `u@v`

Because `u@:v` and `u@v` have very similar definitions, and produce identical results in many cases, almost every beginning J programmer confounds the two. The key is to remember that each sequence produces a *new verb* which has a rank. In `u@:v`, this rank is infinite, so that in `x u@:v y`, the derived verb `u@:v` is applied to the entire `x` and `y`, meaning that `v` is applied to the entire `x` and `y` and `u` is applied to the entire result of `v` . In the other case, the rank of `u@v` is the rank of `v`, so in `x u@v y` the verb `u@v` is applied to individual cells of `x` and `y`, where the cell-size is given by the rank of `v` : for each of those cells, `v` is applied followed by `u`, and the results from the cells are collected into an array.

If we try to take the sum-of-products using `u@v` instead of `u@:v`, we see the difference between the two forms:

```
   1 2 3 +/@* 1 2 3
1 4 9
```

What happened? We thought we were multiplying the vectors and then taking the sum. Because we used `@` rather than `@:`, the derived verb had the rank of dyad `*`, namely 0, which means that the derived verb was applied to each cell: at each cell we multiplied and then took the sum *of the single cell*. In `fndisplay` form,

```
   defverbs 'plus"0 times"0'
   1 2 3 plus/@times 1 2 3
+---------+---------+---------+
|1 times 1|2 times 2|3 times 3|
+---------+---------+---------+
```

`plus` never got executed, because `plus/` was applied to 1-element lists, leaving in each case the single element.

Many J programmers think of `@` and `@:` as establishing a different kind of connection between `u` and `v`, with `u@:v` applying `u` to the entire result of `v` and `u@v` applying `u` to *result cells* of `v` (where a result cell is the output produced by applying `v` to a single operand cell). Such an interpretation makes it easy to understand the operation of `+/@*` : `+/` is applied on result cells of `*`, which are scalars.

The connection interpretation of `u@v` correctly accounts for the results produced by J, but as you use it you should be aware that it is inaccurate because it suggests that `v` is executed against the operand(s) in their entirety. The actual cell-at-a-time execution of `u@v` is different in two ways: it is slower because the verb `v` must be restarted for each cell; and if the temporary space required by `u` or `v` is large, cell-at-a-time execution uses less space because the temporary space for each cell is freed before the next cell is processed.

Making a Monad Into a Dyad: The Verbs [and]

The characters `[` and `]` are not paired in J; each is an independent verb. This is jarring at first but you'll get used to it.

`[` and `]` are identity verbs: they have infinite rank, and `] y` and `[ y` both result in `y` . As dyads, they pick one operand: `x [ y` is `x`, and `x ] y` is `y` . They can be useful when you have a monadic verb that for one reason or another must be used as a dyad with an unwanted operand: then `x v@:[ y` applies `v` to `x`, and `x v@:] y` applies `v` to `y` . Example:

```
   1 2 3 {.@:[ 4 5 6
1
```

Here are some other uses of `[` and `]` . We have already met the first one, which is to display the result of an assignment:

```
   a =. 1 2 3
```

produces no typeout, but

```
   ]a =. 1 2 3
1 2 3
```

Second, `[` can be used to put multiple assignments on the same line, since each application of `[` ignores what is to its right:

```
   a =. 5 [ b =. 'abc' [ c =. 0
```

Finally, `]` can be used instead of parentheses to separate numbers that would otherwise be treated as a list:

```
   5 ,"0 1 2
|syntax error
|   5    ,"0 1 2
   5 ,"0 (1 2)
5 1
5 2
   5 ,"0 ] 1 2
5 1
5 2
```

Making a Dyad Into a Monad: `u&n` and `m&v`

A dyadic verb takes two operands, but if you know you are going to hold one fixed, you can create a monadic verb out of the combination of the dyad and the fixed operand; the monad's operand will be applied to whichever operand of the dyad was not held fixed. The conjunction `&`, when one of its operands is a noun, fixes an operand to a dyad: `m&v y` has infinite rank and is equivalent to `m v y`; `u&n y` has infinite rank and is equivalent to `y u n` . Examples:

```
   2&^ 0 1 2 3
1 2 4 8
   (-&2) 4 5 6
2 3 4
```

Now we can solve our original problem, which was to put **`1.04 * x + y`** into the form **`x v y`** . **`v`** should be

`1.04&* @: +`

which we can verify using **`fndisplay`** as

```
   defverbs 'plus"0 times"0'
   defnouns 'x y'
   x  1.04&times @: plus  y
+------------------+
|1.04 times x plus y|
+------------------+
```

and to get my total savings after receiving all payments I use monad **`/`** to apply that verb for each payment, giving the expression:

`1.04&* @: + / (|.x) , y`

Let's take a moment to understand the parsing and execution of **`1.04&* @: + /`** . **The left operand of a modifier includes all preceding words up to and including the nearest noun or verb that is not immediately preceded by a conjunction**. This is a precise way of saying that modifiers associate left to right. In the phrase **`1.04&* @: + /`**, the **`1.04`** is not preceded by a conjunction, so it is the beginning of all the conjunctions' left operands, and the verb is parsed as if it were written **`(((1.04&*) @: +) / )`** .

Note that left-to-right *association* of modifiers corresponds to right-to-left *execution*. When the derived verb monad **`(((1.04&*) @: +) / )`** is executed, it performs according to the definition of monad **`u/`**, with **`((1.04&*) @: +)`** as the **`u`**; execution of monad **`u/`** inserts **`u`** between items, so **`((1.04&*) @: +)`** is executed between items; at each such execution the dyad **`+`** is executed first, followed by the monad **`1.04&*`** . Fortunately, the result of the parsing rules is that conjunctions and adverbs, just like verbs, should be read right-to-left.

If I wanted to spend half of my uncle's money , the amount I would be left with is

`0.5 * 1.04&* @: + / (|.x) , y`

No parentheses are needed, because **`1.04`** is still not preceded by a conjunction and so **`1.04&* @: + / (|.x),y`** is still evaluated before that value is multiplied by **`0.5`** .

Once you get the hang of it, you will be able to understand and build composite verbs of great power. That will be a useful skill to develop, because you never know when you are going to want to make some sequence of functions the operand of a modifier, and then you're going to have to be able to express the sequence in a single compound verb. It will take a lot of practice, as well as coding techniques to break a gristly mass of conjunction-bound words into digestible pieces (we'll learn them later). For the time being, be content if you can understand the simple examples shown above.

<< >> Contents Help

11. Boxing

(structures)

The nouns we have encountered so far have all had items with identical shapes, and with all atoms of the same type, either numeric or character. You may be afraid that such regular arrays are all that J supports and that you will have to forgo C structures; and you may wonder how J will fare in the rough-and-tumble of the real world where data is not so regular. This chapter will put those fears to rest.

I think we should get a formal understanding of ***boxing*** in J before we try to relate it to structures in C, because the ideas are just different enough to cause confusion. The ***box*** is an *atomic data type* in J, along with number and character. As with the other types, a single box is a scalar, with rank 0 and empty shape. Just as a numeric or character atom has a value, so a boxed atom has a value, called its ***contents***. The box is special in that its contents can be an array while the box itself is an atom. The boxing protects the contents and allows them to be treated as an atom.

Arrays of boxes are allowed, and as always **all the atoms in an array must have the same type**: if any element is boxed, all must be boxed.

Various verbs create boxes. Monad **`<`** has infinite rank and exists for the sole purpose of ***boxing*** its operand: **`< y`** creates a box whose contents are **`y`**, for example:

```
   <1
+-+
|1|
+-+
   <1 2 3
+-----+
|1 2 3|
+-----+
   <'abc'
+---+
|abc|
+---+
```

When a box is displayed, the contents are surrounded by the boxing characters as seen in the examples.

Only certain primitives can accept boxes as operands; generally, you cannot perform arithmetic on boxes but you can do other things like monad and dyad **`#`**, monad and dyad **`$`**, and other primitives that do not perform arithmetic. The significant exception to this rule is that you *can* use monad and dyad **`/:`** and **`\:`** to order boxed arrays. Comparison for equality between two atoms is not strictly an arithmetic operation—you can compare two characters or a character and a number for equality, for example—and it is allowed on boxes, both explicitly using dyad **`=`** and dyad **`-:`** or implicitly using dyad **`i.`** and dyad **`e.`**; but there is an arithmetic flavor to the operation: if the contents of corresponding components of the boxes are both numbers, tolerant comparison is used.

Most primitives that accept boxes as operands do not examine the contents of the boxes, but instead perform their operation on the box atoms themselves. Any deviation from this behavior will be noted in the definition of the verb (we have not encountered any yet). Examples:

```
   3 $ <'abc'
+---+---+---+
|abc|abc|abc|
+---+---+---+
```

The dyad **`$`** was applied to the box, creating a list of identical boxes.

```
   (<1 2),(<5),(<'abc')
+---+-+---+
|1 2|5|abc|
+---+-+---+
```

The boxes were concatenated, resulting in a list of boxes. Note that the contents of the boxes do not have to have the same shape or type.

```
   1 0 1 # (<1 2),(<5),(<'abc')
+---+---+
|1 2|abc|
+---+---+
```

The selection performed by dyad **`#`** is performed on the boxes, not on their contents.

Since applying a verb may result in the addition of verb fills and framing fills, we need to meet the fill element used for boxed nouns. It is the noun **`a:`** . **`a:`** is defined to be **`<0$0`**, i. e. an atom which is a box containing an empty numeric list (note that this is not the same thing as a box containing an empty string or an empty boxed list).

```
   a:
```

```
++
||
++
   3 {. <5
+-+++
|5|||
+-+++
```

`a:` was used for the fills added by overtaking from this boxed noun.

The contents of a box can be any noun, including a box or array of boxes:

```
   < < 2 3
+-----+
|+---+|
||2 3||
|+---+|
+-----+
   (<'abc'),(<<1 2)
+---+-----+
|abc|+---+|
|   ||1 2||
|   |+---+|
+---+-----+
```

The contents of a box can be recovered by ***opening*** the box with monad `>` . Monad `>` has rank 0 (since it operates only on boxes, which are atoms), and its result is the contents of the box:

```
   > < 'abc'
abc
```

The contents of the box is the character string.

```
   a =. (<1 2),(<<5),(<'abc')
   > 0 { a
1 2
   > 1 { a
+-+
|5|
+-+
```

Here we recover the contents of the selected box (which may be a box).

```
   > (<1 2 3),(<4)
1 2 3
```

```
4 0 0
```

Remember that monad **`>`** has rank 0, so it is applied to each box and the results are collected using the frame. Here framing fills were added to the shorter result. **`0`** was used as the framing fill because the contents of the boxes were numeric.

```
   >a
|domain error
|       >a
```

Here the results on the different cells were of different types, so it was impossible to collect them into an array.

If **`y`** is unboxed, the result of **`> y`** is **`y`** .

Terminology

Before we go further with boxes, let's agree on some terminology. Because every atom in an array must have the same type, it is reasonable to speak of an array as being 'boxed' if its atoms are boxed, 'unboxed' otherwise (some writers use 'open' as a synonym for 'unboxed'). Trouble arises with a phrase like 'boxed list'. If I box a list (e. g. **`<1 2 3`**), does that give me a boxed list? If I have a list whose atoms are boxes, (e. g. **`(<1),(<2)`**), is that also a boxed list? Unfortunately, writers on J have not agreed on terminology for these cases.

In this book, a ***boxed list*** will be a list that has been put into a box (it is therefore an atom), and a ***list of boxes*** is a list each atom of which is a box. Higher ranks are described similarly. If we say that a noun '***is boxed***', that simply means that its atoms are boxes.

Boxing As an Equivalent For Structures In C

A C **struct** corresponds to a list of boxes in J, where each box in the list corresponds to one structure element. **Referencing a structure element** in C corresponds to selecting and opening an item in J. For example,

```
struct {
    int f[ ] = {1,2,3};
    char g[ ] = "abc";
    float h = 1.0;
)
```

is equivalent to

```
(<1 2 3) , (<'abc') , (<1.0)
```

A C **n-dimensional array of structures** is equivalent to a J array of boxes of rank n+1. The last axis of the J array corresponds to the structure, and its length is the number of structure elements in the C structure.

C has no exact equivalent for a single box.

<< >> Contents Help

12. Empty Operands

Every programmer knows that errors lurk near boundaries: for example, at the beginning and end of arrays, or at the point in execution where an array becomes empty. Our discussion of verb rank omitted one important situation: suppose there are no cells? An operand with no cells is said to be ***empty***, and it gets special treatment in J.

The definition of an empty operand is not as obvious as you might think. An *array* is empty if it has no atoms (i. e. has a **`0`** in its shape), but whether an *operand* is empty depends on the rank of its cells: it is empty if there is a **`0`** in the frame **with respect to the cells operated on by the verb**. Consider an array with shape **`3 0 4`** . This is an empty array, because it has no atoms. As an operand of a verb with rank 0, it has frame **`3 0 4`**, so there are no cells and the array is an empty operand. As an operand of a verb with rank 1, it has frame **`3 0`** and again is empty. As an operand of a verb with rank 2, though, it has frame **`3`** and is not empty: there are 3 cells, each of which has shape **`0 4`** . In this case each *cell* is an empty array, but the *operand* is not empty. As an operand of a verb with rank 3 or higher, the frame is empty and each cell has shape **`3 0 4`**, so there is one cell and the operand is not empty, though each cell is an empty array. You can see that an operand may have items and yet be an empty operand, if the verb operates on cells smaller than items, as is the case in this example for a verb of rank 0 or 1.

We are left with the question, What is executed when an operand is empty? For *something* must be executed. It is fundamental in J that if a verb produces a result with shape *s* when applied to a single cell, executing that verb over an array of that cell—even an array of none of them—produces an array of results each with shape *s*. The only way to find out what shape a verb is going to produce is to execute it and see—and that is what J does.

Execution On a Cell Of Fills

If an operand is empty, i. e. its frame contains a **`0`** and it therefore has no cells, the verb is executed on a cell *c* of fills. The shape of *c* is the shape of a cell of the corresponding operand, and the value of each atom is the appropriate fill for that operand: **`0`** for numeric operands, **`' '`** for characters, **`a:`** for boxes. The shape *s* and type *t* of the result of executing the verb on *c* are noted. Then, the shape of the overall result is the frame of the operand (the longer frame, for dyads) concatenated with *s*, and the result is given type *t*. The result will necessarily be empty, because it will have a `0` in its shape (from the frame, which contained a `0`). Example:

```
   $ +/"2 (3 0 3 4 $ 100)
3 0 4
```

Remember that a cell has a shape, even if there are none of them! Here the verb monad `+/"2` is applied to 2-cells, each with shape `3 4` . The frame `3 0` contains a `0`, so the verb is executed on the fill cell *c* which is (`3 4 $ 0`). Monad `+/` adds the items of the cell, producing a list with shape `4` . The frame `3 0` is concatenated with `4` to give the shape of the result, `3 0 4` .

```
   3!:0 +/"2 (3 0 3 4 $ 100)
4
```

The verb `3!:0` (one of dozens of special goodies provided by the `!:` conjunction and documented under Foreigns) tells you the type of its operand. Here, `4` means numeric type: the result has shape `3 0 4` and numeric type.

```
   $ <"2 (3 0 3 4 $ 100)
3 0
   3!:0 <"2 (3 0 3 4 $ 100)
32
```

Here the verb monad `<` was applied to the same fill-cell *c* (`3 4 $ 0`) but it produced a boxed scalar result (shape empty), so the shape of the overall result is the frame `3 0` concatenated with an empty list, i. e. shape `3 0` and type boxed (as indicated by the `32` returned by `3!:0`).

Note that the contents of each box in an empty array of boxes are all empty. In the example `<"2 (3 0 3 4 $ 100)`, execution on the fill-cell produced (`<3 4 $ 0`), and if the frame weren't empty the box would contain an array; but when the frame is empty, the value of the result is discarded; all that remains is its type.

If executing the verb on a cell of fills results in an error, execution continues as if the verb had returned the scalar `0` :

```
   5 + ' '
|domain error
|   5    +' '
```

Trying to add 5 to a space is nonsense…

```
   5 + ''
   $ 5 + ''
0
   (3!:0) 5 + ''
4
```

…but adding 5 to an empty list of characters produces an empty numeric list. The addition is attempted with a cell of fills for the empty operand (the values added are **`5`** on the left, **`' '`** on the right), the addition fails, and the error-fallback result of scalar **`0`** is used; appending the longer frame (**`0`**) gives shape **`0`**, numeric. Note that **`y`** here is an empty operand, but it nonetheless has the longer frame (the scalar **`x`** has empty shape and perforce empty frame).

Note: there is much special code in the interpreter to handle the cases of empty operands. To improve performance, the interpreter recognizes a great many combinations of verb, operand shape, and type, handling each with separate code. In most cases the interpreter produces its result in accordance with the rules given above, but in a few exotic cases it deviates. You are quite unlikely to encounter these cases in practice; the most important one is

```
   $ > 0$a:
0
```

where the rules given above would predict a shape of **`0 0`** . Enough applications rely on shape **`0`** to keep this deviation in the system, at least as of release 5.01.

Empty cells

As we discussed above, a cell, like any array, is called empty if it has a **`0`** in its shape. Whether the cells of an operand are empty is independent of whether the operand itself is empty.

How a verb handles an empty cell is entirely up to the verb; the fill-cell processing we discussed above does not apply. The J primitives generally preserve the type of empty lists that are 'data' but ignore the type of empty lists that are 'control information'. So, even though characters are not allowable left operands of dyad **`|.`**, **`'' |. i. 5`** produces the same result as **`(0$0) |. i. 5`**, because the rotation count is 'control information'. In contrast, **`3 {. ''`** produces a 3-character string, while **`3 {. (0$0)`** produces a 3-item numeric list, because the right operand of dyad **`{.`** is 'data'. The distinction between 'control information' and 'data' is not clear-cut, but in all cases the interpreter does what you'd want it to, and you should experiment if you need reassurance.

If Fill-Cells Are Not Enough

Sometimes executing a verb on a cell of fills simply won't do: maybe your verb produces a side effect, or maybe it will go berserk if its operand is `0` . In those cases, you must take steps to ensure that it is not executed on an empty list. To help you out with the most common case, in which the only way a list can be empty is to have no items (that is another way of saying that the first item of the shape is `0`), I offer you a set of adverbs and conjunctions which you can have by executing `load 'system\packages\misc\jforc.ijs'`

`u Ifany` applies the verb `u` (which may be monadic or dyadic) provided `y` has items; if `y` has 0 items, the result of `u Ifany` is `y` :

```
   $ (,/) i. 0 4
0
   $ (,/) Ifany i. 0 4
0 4
```

Since `i. 0 4` has no items, `Ifany` caused it to be passed through unchanged.

`x u Ifanyx y` produces `x u y` if `x` has items, or `y` if `x` has no items.

The conjunction `u Butifnull n` can be monadic or dyadic; it applies `u` if `y` has items; if `y` has no items it produces a result of `n` .

```
   5 + Butifnull 6 (0)
5
   5 + Butifnull 6 (0$0)
6
```

`x u Butifxnull n y` produces `x u y` if `x` has items, or `n` if `x` has no items.

<<   >>   Contents   Help

13. Loopless Code III—Adverbs `\` and `\.`

We have learned about ordinary verbs that operate on the cells of an operand, and we have learned `u/` which operates between all the cells of its operand. In between those extremes are verbs that operate on subsets of the cells of its operand. In this chapter we will learn a couple of adverbs that apply verbs to subsets of cells chosen according to simple rules; in later chapters we will learn how to form irregular subsets.

`u\ y` has infinite rank and applies `u` to successive ***prefixes*** of `y` . It applies `u` to the first item of `y` (i. e. `u 1 {. y`) to produce the first item of the result; it then applies `u` to the first 2 items of `y` (i. e. `u 2 {. y`) to produce the second item of the result; and so on, with the last item of the result being `u (#y) {. y` . Example:

```
   #\ 9 8 7 6
1 2 3 4
```

This just gives the length of each prefix, not a terribly edifying result. The details can be seen using `fndisplay`:

```
   defverbs 'tally'
   tally\ 9 8 7 6
+-------+---------+-----------+-------------+
|tally 9|tally 9 8|tally 9 8 7|tally 9 8 7 6|
+-------+---------+-----------+-------------+
```

Note that `u` is always applied to lists, because `{.` always produces a list. In particular, the first item of the result comes from applying `u` to a 1-item list; conversely, even if `y` is a scalar, `(#y) {. y` is a 1-item list. Note also that if the applications of `u` produce results of different shapes, framing fills are added to bring the results to a common shape, just as if they were the results from applying a verb to different cells:

```
   ]\ i. 3
0 0 0
0 1 0
0 1 2
```

The result is the prefixes themselves, assembled as items of a rank-2 array.

`u\` is most often used when `u` is of the form `u/`, i. e. as `u/\` . Then the verb `u/` is applied to the successive prefixes. Here are some commonly-used forms:

```
   +/\ i. 6
0 1 3 6 10 15
```

`+/` means 'total the items', so `+/\ i. 6` is `(0),(0+1),(0+1+2),(0+1+2+3)` **…**, i. e. the running total of the items of `y` .

```
   >./\ 9 5 3 10 3 2 20
9 9 9 10 10 10 20
```

For each item of `y`, the result is the largest item occurring in the list up to that item of `y` .

```
   </\ 0 0 0 0 1 1 0 0 1 0
0 0 0 0 1 0 0 0 0 0
```

`</\ y` on a Boolean list (i. e. one containing only `0` or `1`) is a tricky way to turn off all `1`s following the first. See how it works: `</ y` will produce a result of `1` only in the case where the last item of `y` is `1` and the rest are `0`, i. e. `0 < 0…0 < 1`, so `</\ y` produces a `1` for that prefix and `0` for all the others.

```
   *./\ 1 1 1 1 0 0 1 1 1
1 1 1 1 0 0 0 0 0
```

Keep the leading `1`s of `y`, but set the rest to `0` .

`u\. y` is similar to `u\ y`, except that it applies `u` to ***suffixes*** of `y` . It applies `u` to all of `y` (i. e. `u 0 }. y`) to produce the first item of the result; it then applies `u` to all but the first item of `y` (i. e. `u 1 }. y`) to produce the second item of the result; and so on, with the last item of the result being `u ((#y)-1) }. y` (that is, `u` applied to the last item of `y`). Example:

```
   +/\. i. 6
15 15 14 12 9 5
```

The running total now starts at the end and runs backward. `fndisplay` shows the details, and points out a subtlety of `u/\` :

```
   defverbs 'plus"0'
   plus/\. <"0 i. 4
+---------------------+--------------+--------+-+
|0 plus 1 plus 2 plus 3|1 plus 2 plus 3|2 plus 3|3|
+---------------------+--------------+--------+-+
```

Up till now we have applied unboxed inputs to verbs defined by `defverbs` and gotten useful results. Why then must we box the atoms of `i. 4` before giving them to `plus/\.`? The reason is that the result of a verb defined by `defverbs` is boxed. Normally it is joined in an array with other boxed outputs. Here, the last result, the one containing just the cell `3`, is not produced by `plus`; rather, since `plus/` is applied to a 1-element list, the cell is the **unmodified input cell**. If we had not boxed the atoms of `i. 4` (i. e. if we had executed `plus/\. i. 4`), the unboxed `3` would be joined to the other boxed results, and that would have given us a domain error. Whenever you use `u/\ y` or `u/\. y` you must make sure that the result of `u` has the same type (character, numeric, or boxed) as `y` .

The dyadic form `x u\ y` has rank `0 _` and applies `u` to ***infixes*** of `y` . An infix is a sequence of adjacent items. `x` gives the length of the infixes. The first item of the result comes from the infix of length `|x` (that is, the absolute value of `x`) starting with the first item of `y`, and subsequent items of the result come from subsequent infixes. If `x` is positive, successive infixes start at successive items of `y` (therefore, they overlap), and the last infix is the one that ends with the last item of `y`; if `x` is negative, infixes do not overlap: each one starts with the item following the last item of the previous infix, and the last infix may be shorter than `|x` . Examples:

```
   _2 ]\ 100 2 110 6 120 8 130 3
100 2
110 6
120 8
130 3
```

This is a convenient way to reshape a list to have 2 items per row when you don't know how many rows there will be.

```
   2 -/\ 10 8 6 4 2
2 2 2 2
   2 -~/\ 10 8 6 4 2
_2 _2 _2 _2
```

Applying `-/` between each pair of items (with adjacent pairs overlapping) takes the backward difference of each pair. To take the forward difference, subtract the first from the second using `-~/` .

```
   3 >./\ 1 2 3 8 2 3 1 5 4 3 12 3 2
3 8 8 8 3 5 5 5 12 12 12
```

This takes the maximum over a rolling window 3 items wide.

`x u\. y` is similar to `x u\ y` , but it operates on ***outfixes*** which contain all of `y` except the corresponding infix. Its uses are few.

The interpreter treats empty operands with care, so that you don't have to worry about them as special cases. If you simply must know the details, here they are: If **`u\`** or **`u\.`** is applied where there are no applicable subsets of **`y`** (either because **`y`** is empty or because it is too short to muster even a single infix), **`u`** is applied to a list of fills **`f`** and the result has 0 items, each with the shape and type of the result of **`u f`** . The items of **`f`** have the shape of items of **`y`**, and the length of **`f`** is **`0`** for monad **`u\`** or **`u\.`**, or the length of an infix or outfix for dyad **`u\`** or **`u\.`** . For example:

```
   $ 2 ]\ i. 1 3
0 2 3
```

We were looking for infixes of length 2; each one would have had shape **`2 3`**, but with only one item in **`y`** we have insufficient data for an infix. So, we create a 2×3 array of fills and apply the verb **`]`** to it; the verb returns shape **`2 3`** and the overall result is a list of 0 items with that shape.

```
   $ ]\ i. 0 2 3
0 0 2 3
```

The cells of **`y`** have shape **`2 3`** so we apply the verb **`]`** to a list of 0 of them (i. e. with shape **`0 2 3`**). The result of **`]`** has the same shape as the input, and the final result is a list of 0 items each with shape **`0 2 3`**, i. e. with overall shape **`0 0 2 3`** .

---

14. Verbs for Arithmetic

It's time to build your J vocabulary with a few more verbs.

Dyads

`x ! y` (rank **`0 0`**) number of ways to choose **`x`** things from a population of **`y`** things. More generally, **`(!y) % (!x) * (!y-x)`**

`x %: y` (rank **`0 0`**) The **`x`**th root of **`y`**

`x #. y` (rank **`1 1`**) **`y`** evaluated in base **`x`**, i. e. the atoms of **`y`** are digits in the base-**`x`** representation. **`x`** and **`y`** must have the same length, unless **`x`** is a scalar in which case it is replicated to the length of **`y`** . A place-value list **`p`** is calculated, in which each item is the product of all the subsequent corresponding items of **`x`**; formally, this is **`p=.*/\.}.x,1`** (the last item of **`p`** is always 1 and the first item of **`x`** is immaterial). Then each item in **`y`** is multiplied by its place value and the results are summed to give the result (formally this is **`+/ y * p`**). In the simplest case **`x`** is a scalar:

```
   10 #. 2 3 4
234
```

The digits **`2 3 4`** interpreted as base-ten digits.

```
   16 #. 2 3 4
564
```

The same interpreted as base-sixteen digits.

```
   16b234
564
```

as expected (**`16b234`** is 234 in base 16, equivalent to **`0x234`**).

Here is an example where **`x`** is a list, converting time in hours/minutes/seconds to seconds since midnight:

```
   24 60 60 #. 12 30 0
45000
```

The list **`p`** was **`3600 60 1`** . The first item in **`x`** has no effect on the result.

`x #: y` (rank **`1 0`**) This is the inverse of **`x #. y`** except that if the first digit of the result is not between **`0`** and **`{.x`**, it is changed (modulo **`x`**) to lie within that range. For example,

```
   24 60 60 #: 45000
12 30 0
```

The normal case, converting the number of seconds since midnight back to hours/minutes/seconds.

```
   24 60 60 #. 36 30 0
131400
   24 60 60 #: 131400
12 30 0
```

Here the result would have been **`36 30 0`**, but **`36`** is outside the range **`0`** to **`24`**, so it is replaced by **`24|36`**, which is **`12`** . If you want the first item of the result to be unconstrained, make the first item of **`x`** either **`0`** or _ :

```
   0 60 60 #: 131400
36 30 0
```

Note that monad **`#:`** and monad **`#.`** are similar to the dyadic forms with **`x`** set to **`2`** . To keep **`#:`** and **`#.`** straight in your mind, remember that the one with a single dot (**`#.`**) produces a single scalar result, while the one with multiple dots (**`#:`**) produces a list result.

Monads (all rank 0)

`! y` factorial of **`y`** (more generally, the gamma function Γ(1+**`y`**))

`^ y` same as (the base of natural logarithms *e*) **`^ y`**

`^. y` same as (the base of natural logarithms *e*) **`^. y`**

`+: y` same as **`y * 2`**

`-: y` same as **`y % 2`**

`*: y` same as **`y ^ 2`**

`%: y` same as **`2 %: y`**

You can see the common feature in the arithmetic primitives ending in **`:`** .

15. Loopless Code IV

In our quest to write loopless code, we first learned about J's implicit looping, which we can use to replace loops in which the same function is performed on each cell; then we learned monad **`/`** which lets us accumulate an operation across all the items of a noun, and **`\`** and **`\.`** which apply verbs to certain regular subsets of a noun. We now examine cases in which the operations on the cells are different, but where there is no sharing of information between cells.

A Few J Tricks

In these irregular cases, the J solution is to create an array that contains control information describing the difference between the cells, and then create a dyadic operation that produces the desired result when given a cell of control information and a cell of data. Writing code this way can seem ingeniously clever or awkwardly roundabout, depending on your point of view; we will simply accept it as a necessary part of coding in J, and we will learn to be effective with it. What follows is a hodgepodge of tricks to treat cells individually. If we were writing in C, we would use **if** statements, but since **if** by necessity involves a scalar comparison we will avoid it in J.

To add one to the elements of **`y`** whose values are even:

```
y + 0 = 2 | y
```

To double all the elements of **`y`** whose values are even:

```
y * 1 + 0 = 2 | y
```

To create an array whose even-numbered elements come from **`y`** and whose odd-numbered elements come from **`x`** :

```
(x * -.sv) + y * sv =. (#y) $ 1 0
```

which, homely as it is, is a standard idiom in J. This expression works only for numeric operands; for general operands we can select using a selection vector **`sv`** with

```
sv {"_1 x ,. y
```

To replace lowercase 'a' through 'f' with uppercase 'A' through 'F' in a string that contains only 'a' through 'f':

```
('abcdef' i. y) { 'ABCDEF'
```

Extending the previous example: to replace lowercase 'a' through 'f' with uppercase 'A' through 'F' leaving other characters unchanged:

```
(('abcdef' , a.) i. y) { 'ABCDEF' , a.
```

To understand this you need to know the special noun **a.** which is the character string containing all the ASCII characters in order. Work through a simple example until you understand how this works—it's a good example of how J thinking differs from C thinking.

A similar problem: given a list of keys **y** and a list of data **z**, with each item of **y** corresponding to an item of **z**; and another list of search keys **x**; and a default element **d** : return the item in **z** corresponding to the item of **y** that matches the item of **x**, or **d** if the item of **x** didn't match anything in **y** :

```
(y i. x) { z , d
```

To evaluate the polynomial defined by **x**, so that if for example **x** is **2 1 5** the result is $5y^2+y+1$:

```
+/ x * y ^ i. # x
```

(and now you can see why **0^0** is **1**).

To evaluate the polynomial defined by **x** going the other direction, so that if for example **x** is **2 1 5** the result is $2y^2+y+5$:

```
y #. x
```

The last example, due to Roger Hui, has a power and economy that amount to sorcery. Suppose you had a list, and you wanted to know, for each item in the list, how many identical items appeared earlier in the list. You could find out this way:

```
   y =. 2 2 2 1 2 1 4 6 4 2
   t - (i.~ y) { t =. /: /: y
0 1 2 0 3 1 0 0 1 4
```

Take a little time—maybe a long time—to see how this works. The **/: /: y** is an idiom we discussed earlier—did you figure it out? It gives the *ordinal* of each item of **y**, in other words the rank of the item among the items of **y** . If there are equal items, they will occupy a block of successive ordinals. In this example you can see that **t** does indeed hold the ordinals:

```
   t
2 3 4 0 5 1 7 9 8 6
```

(i.~ y) takes the index of each item of **y** within **y** itself, in other words, for each item, the index of the first item with the same value:

```
   (i.~ y)
```

```
0 0 0 3 0 3 6 7 6 0
```

Since the identical items of **`y`** are a block of successive ordinals, and **`(i.~ y)`** comprises indexes of first items in blocks, we can find relative positions in blocks by subtracting the ordinal of the first item with a value from the ordinals of all the other items with the same value. That is what this expression does. Lovely!

In addition to the foregoing *ad hoc* means of varying the operation cell-by-cell J has some language features expressly designed for that purpose:

Power/If/DoWhile Conjunction `u^:n` and `u^:v`

`u^:n y` has infinite rank. It applies the verb **`u`** to **`y`**, then applies **`u`** to that result, and so on, for a total of **`n`** applications of **`u`**, ; in other words **`u u u…`** **`(`*`n times`*`) y`**, as we see when it is used with the **`>:`** (increment) primitive:

```
   >: 5
6
   >:^:2 (5)
7
   >:^:3 (5)
8
```

`fndisplay` gives a picture of what is happening:

```
   defverbs 'incr"0'
   incr^:3 (5)
+----------------+
|incr incr incr 5|
+----------------+
```

`x u^:n y` also has infinite rank. It evaluates **`x u x u…(`*`n times`*`) y`** . A simpler way to say this is to say that it is equivalent to **`x&u^:n y`**, since **`x&u y`** is equivalent to **`x u y`** .

```
   2 * 2 * 2 * 2 * 5
80
   2 *^:4 (5)
80
```

`u^:v y` and **`x u^:v y`** are defined similarly: first **`v`** is evaluated (monadically or dyadically as appropriate), and then result is used as **`n`** . Formally, **`u^:v y`** is **`u^:(v y) y`** and **`x u^:v y`** is **`x u^:(x v y) y`** . With dyad **`u^:v`**, it will be rare that **`x`** and **`y`** both make sense as an operand into both **`u`** and **`v`**, and you will usually use `@:[` and `@:]` to cause `u` or `v` to operate on only one operand. For example, to coalesce the `x+1` leading axes of `y` into one axis, you could use `x ,/@:]^:[ y` :

```
   1 ,/@:]^:[ i. 2 2 3
0  1  2
3  4  5
6  7  8
9 10 11
   2 ,/@:]^:[ i. 2 2 3
0 1 2 3 4 5 6 7 8 9 10 11
```

This is hardly a commonplace usage, but let's analyze it, since conjunctions are still new to us. The verb is parenthesized `((,/)@:])^:[`, so the first thing executed is `u^:v` where `v` is `[` . `x [ y` is just `x`, so `x` is going to tell us how many times to apply `x&((,/)@:])` . Now, `x&((,/)@:]) y` is just the same as `,/ y`, because the purpose of the `@:]` is to ignore the left argument that was put on by `x&` . We remember monad `,/` from our discussion of monad `u/` : it combines the 2 leading axes of `y` . So, `x ,/@:]^:[ y` will combine the 2 leading axes of `y`, `x` times; in other words, combine the `x+1` leading axes of `y` .

Our interest in `u^:n` is not in applying a verb several times—usually we could just write the instances out if we needed to—but rather in 4 special values of `n` : `_1`, `0`, `1`, and _ (infinity). `u^:0`, meaning apply `u` 0 times, is simple: it does nothing, with the result `y` in both dyadic and monadic forms. `u^:1` means apply `u` once. Thinking about that, we see that if `n` is restricted to the values 0 or 1, `^:n` means `'If n'` : `u^:n y` is `y`, but modified by application of `u` if `n` is 1. If we want to apply the verb `u` only on the items of `y` for which `x` is 1, we can write `x u@:]^:["_1 y` :

```
   1 0 0 1 0 >:@]^:["_1 (1 2 3 4 5)
2 2 3 5 5
```

When `n` is _, `u^:_` means 'keep applying `u` until the result doesn't change'. This is obviously a splendid way to perform a numerical calculation that converges on a result; for example if you take …*cos(cos(cos(cos(y))))* until the result stops changing, you get the solution of the equation *y=cos(y)*:

```
   2 o.^:_ (0)
0.739085
```

but that's not why we love `^:_` . Consider the verb `u^:v^:_` (either monad or dyad), with the understanding that `v` always produces a Boolean result of `0` or `1` . It is parenthesized `(u^:v)^:_`, i. e. `u^:v` repeated until the result stops changing. Now, if `v` evaluates to `0`, the result of `u^:v` will certainly be the same as its input because `u` will not be executed; if `v` is 1, `u^:v` causes `u` to be executed once. So this construct is like C's **while(v(y))y = u(y);** (except that the J version also stops if the result of `u y` is the same as `y`). The great thing about having a *verb* to do this loop rather than a *statement* is that we can give the verb a rank and apply it to cells, with independent loop control on each cell:

```
   2 *^:(100&>@:])^:_"0 (1 3 5 7 9 11)
128 192 160 112 144 176
```

Read this as 'for each atom of `y`, double it as long as the value is less than 100'.

`u^:_1` is also of great interest but we will discuss it later.

One last point:

```
   >:^:1 2 4 (5)
6 7 9
```

As you can see, `n` may be an array, in which case `u^:n1 y` is repeatedly evaluated, with `n1` assuming the value of each atom of `n`, and the results are assembled using the shape of `n` as the frame with framing fills added as needed. Pop quiz: express the preceding English sentence in J.

Solution: `u^:n y` is equivalent to `n u@:]^:["0 _ y`, and `x u^:n y` is equivalent to `n x&u@:]^:["0 _ y` . If you can make sense of the answer, you should be content with your progress. If you came up with either half on your own, you are entitled to claim Apprentice Guru status.

Tie and Agenda (switch)

The Tie Conjunction `` u`v u`n m`v m`n ``

The backquote character `` ` `` is the conjunction named `Tie`. `` ` `` is one of the few conjunctions that produce a noun, so it is neither monadic or dyadic. If an operand of `` ` `` is a verb, it is converted to its ***atomic representation*** which is a noun form from which the verb can be recovered; then the two operands `m` and `n` (both nouns now since any verb was converted to a noun) are joined by executing `m,n` . So, the result of `` ` `` applied between the members of a sequence of verbs is a list of special nouns, each of which is the atomic representation of a verb. We are not concerned with the format of the atomic representation, nor will we create or modify an atomic representation (that's Advanced-Guru work); we will be content to use the values produced by ` . An example is:

```
   +`-`*`%`(+/)
+-+-+-+-+-------+
|+|-|*|%|+-+---+|
| | | | ||/|+-+||
| | | | || ||+|||
| | | | || |+-+||
| | | | |+-+---+|
+-+-+-+-+-------+
```

What makes the result of ` special is not the boxing, but the fact that what's in the boxes is not just any old data, but data in the format that can be used to recover the original verbs. Once created, the result of ` can be operated on like any other noun:

```
   a =. +:`-`*`%`(+/)
   3 { a
+-+
|%|
+-+
   0 0 1 0 1 # a
+-+-------+
|*|+-+---+|
| ||/|+-+||
| || ||+|||
| || |+-+||
| |+-+---+|
+-+-------+
```

In English grammar, a gerund is a form of a verb that is used as a noun, for example the word *cooking* in *Cooking is fun*. The result of ` in J is also called a ***gerund***, and we can see that the name is apt: a gerund in J is a set of J verbs put into a form that can be used as a J noun. It has the latent power of the verbs put into a portable form, like nitroglycerine that has been stabilized by kieselguhr to become dynamite. The blasting cap that sets it off is

The Agenda (switch) conjunction `m@.v`

`m@.v` (either monad or dyad) uses the result of `v` to select a verb from the list of verbs `m`, and then executes that verb.

`m@.v` requires that `m` be a valid gerund. It produces a verb which can be used monadically or dyadically and whose ranks are the ranks of `v` . The operation of this verb is as follows: `v y` (if monadic) or `x v y` (if dyadic) is evaluated; it must produce a scalar result `r` that is a valid index into `m`; i. e. `(-#m) <: r` and `r < #m` . Then, item `r{m` is selected—it is the atomic representation of one of the verbs that went into `m`—and that atomic representation is converted to a verb `u` . Finally, `u y` (if monadic) or `x u y` (if dyadic) is executed, and its result is the result of the execution of `m@.v` .

So, `` verb0`verb1`verb2 @. v y `` evaluates `v y`, resulting in *r*, and then executes `verb`*r* `y` . The dyadic case `` x verb0`verb1`verb2 @. v y `` evaluates `x v y`, resulting in *r*, and then executes `x verb`*r* `y` . The verbs may be any valid verb: a primitive, a compound verb, or a named verb.

Examples:

```
   (1&+)`(-&2)@.(2&|) "0 i. 6
1 _1 3 1 5 3
```

This added 1 to each even number and subtracted 2 from each odd number. Note that we had to assign rank 0 to the overall combined verb, because otherwise the rank of `` (1&+)`(-&2)@.(2&|) `` would have been the rank of `2&|` which is infinite because `m&v` has infinite rank.

```
   _5 _3 _1 1 3 5 +`-@.(0&>@:["0) 2
_7 _5 _3 3 5 7
```

Subtract 2 from elements of `x` that are negative, add 2 to elements that are nonnegative. Here we assigned the rank to the selector verb in `m@.v`; that rank was then inherited by `m@.v` .

```
   5 uname`+`] @. (*@:]"0) _5 0 5
```

(Remember that monad `*` is the signum function returning `_1` for negative, `0` for zero, and `1` for positive operands) For each atom of `y`, execute `5 uname y` if `y` is zero, `5 + y` if `y` is positive, and pass `y` through unchanged (`5 ] y`) if `y` is negative. `uname` must be defined elsewhere. This expression makes use of negative indexing: if `* y` is negative, verb number `_1` (the last item) is taken from the gerund.

`m@.v` obviously can be used with a small rank to afford great control over what operation is performed cell-by-cell, but if you do that it will have to apply J verbs on small operands, which is inefficient. After all we've been through, I feel confident that I can trust you not to use **`m@.v`** with small rank unless it's absolutely necessary.

<< >> Contents Help

16. More Verbs For Boxes

Dyad `;` (Link) And Monad `;` (Raze)

Dyad **`;`** has infinite rank. It boxes its operands and concatenates the boxes:

```
   'abc' ; 1 2 3
+---+-----+
|abc|1 2 3|
+---+-----+
```

Dyad **`;`** is the easiest way to create a list of boxes:

```
   'abc' ; 1 2 3 ; (i. 2 2)
+---+-----+---+
|abc|1 2 3|0 1|
|   |     |2 3|
+---+-----+---+
```

Did you notice that I gave you an inaccurate definition? If dyad **`;`** just boxed the operands and concatenated, its result would be like

```
   dyadsemicolon =: dyad : '(<x.) , (<y.)'
   'abc' dyadsemicolon 1 2 3 dyadsemicolon (i. 2 2)
+---+-----------+
|abc|+-----+---+|
|   ||1 2 3|0 1||
|   ||     |2 3||
|   |+-----+---+|
+---+-----------+
```

That's not the list of boxes we wanted! Actually dyad **`;`** is more subtle: **`x ; y`** always boxes **`x`**, but it boxes **`y`** *only if* **`y`** *is unboxed*. That produces the behavior we want most of the time; the exception is when the last item in the list is boxed already:

```
   (<'abc');(<'def');(<'ghi')
+-----+-----+---+
|+---+|+---+|ghi|
||abc|||def||   |
|+---+|+---+|   |
+-----+-----+---+
```

If we expected all the items to be boxed the same, we are disappointed. We must develop the habit that when we use a sequence of dyad `;`s we box the last operand (unless we are sure it is unboxed, and even then we might do it to reinforce our habit):

```
   (<'abc');(<'def');<(<'ghi')
+-----+-----+-----+
|+---+|+---+|+---+|
||abc|||def|||ghi||
|+---+|+---+|+---+|
+-----+-----+-----+
```

Monad `;` (infinite rank) removes one level of boxing from an array of boxes, concatenating the contents of all the boxes into one long list. The shape of the operand of `;` is immaterial. If the items do not have a common shape—the *items* of the *contents* of the operand boxes, mind you, not the contents themselves—they are brought up to a common shape as described below. Examples:

```
   'abc';'d';'ef'
+---+-+--+
|abc|d|ef|
+---+-+--+
```

A list of boxes.

```
   ; 'abc';'d';'ef'
abcdef
```

The items, which are scalars, are joined into one long list.

```
   ; 1 ; 2 3 4
1 2 3 4
```

It works for numbers too.

```
   ; (i. 2 3) ; (i. 2 3)
0 1 2
3 4 5
0 1 2
3 4 5
```

The items are lists, so the lists are concatenated into a rank-2 array.

```
   ; 1 2 ; i. 2 3
1 2 0
0 1 2
```

```
3 4 5
```

Here the second box contains a rank-2 array, while the first box contains a rank-1 array. The *items* of highest rank have rank 1, which means that rank-1 items are going to be lined up as the items of a rank-2 result. Any contents of lower rank are brought up to the rank of the highest-rank contents, by adding single-length leading axes; after that operation, the items of all the modified contents have the same rank (but not necessarily the same shape). If the shapes of any of those items differ, verb fills are added to bring each axis up to the length of the longest axis; then the items are assembled into a list whose rank is 1 higher than the rank of an item. In this example the concatenated items have rank 1, and verb fill was added to bring the single item of the first box up to a length of 3.

```
   ; 1 ; ,: 2 3 4
1 1 1
2 3 4
```

There is one amendment to the processing as described above: if any of the contents is an atom, it is *replicated* to bring it up to the shape of an item of the result before items are concatenated. Here the atom `1` was replicated to become `1 1 1` .

```
   ; (,1) ; ,: 2 3 4
1 0 0
2 3 4
```

Here the first box was a 1-item list rather than an atom, so it was padded with fills rather than replicated.

When you have an array of boxes, the difference between opening it with monad `>` and with monad `;` is that monad `>` keeps the frame of the array of boxes, and brings every opened box up to the same shape, while monad `;` just runs the items of the contents together into one long list with no regard for the shape of the array of boxes.

Dyad `,` Revisited—the Case of Dissimilar Items

When we discussed dyad `,` we glossed over the treatment of operands with items of different shapes. Now we can reveal that the padding and replication for dyad `,` is just like what monad `;` does on the contents of boxes. In fact, `x , y` is equivalent to `; (<x),(<y)` .

Verbs With More Than 2 Operands—Multiple Assignment

Dyad `;` is part of the standard J method of passing many operands to a verb. The invocation of the verb normally looks like this:

```
   verbname op1 ; op2 … ;< opn
```

(the `<` is needed only if `opn` is boxed), and the verb that is so invoked looks like:

```
verbname =: monad define
'op1 op2 … opn' =. y.
remainder of verb
)
```

The line `'op1 op2 … opn' =. y.` is J's handy ***multiple assignment***. When the target of the assignment is a string, the string is broken into words and the words are matched with items of the value being assigned (they must match one-for-one or a length error results). Then, each word from the string is used as a name which is assigned the corresponding item of the value. If the value being assigned is boxed, each item is unboxed before it is assigned.

When defined and invoked as shown above, the variables `op1`, `op2`, etc. during execution of the called verb will hold the values of `op1`, `op2`, etc. in the calling verb.

Multiple assignment is not restricted to parameter-passing; you may use it as you see fit, if only to save typing. I have found it very useful in loading configuration parameters from a file: the file contains both noun-names and the values, with the values being assigned to the names by multiple assignment. Such a design is easily portable from release to release of a product, since the file has no 'format'—it simply defines all the names it understands.

A multiple assignment can produce verbs and modifiers as well as nouns. You put a `` '`' `` character before your list of names:

```
   '`name1 name2…' =. list of atomic representations
```

and each `name` is assigned the entity described by the `atomic representation`. Each atomic representation is a noun, but it may describe any kind of entity. Usually your entities will be verbs, and then they can be converted to atomic representations by `` ` ``, as in

```
   '`add subtract mult div' =. +`-`*`%
```

Dyad { Revisited

Now that we know about boxes, we can understand the full description of the selection verb dyad `{` . In the general form, the left argument of `x { y` is a box whose contents is a list of boxes. Pictorially, it is

```
+----------------------------------------+
|+----------------+----------------+-+|
||axis-0 selections|axis-1 selections|…||
|+----------------+----------------+-+|
+-------------------------------------+
```

We will call the inner boxes (i. e. the items of the contents of the box `x`) the ***selectors***. The first selector gives the indexes to be selected along the first axis (i. e. axis 0); the second selector gives the selections for axis 1; and so on.

```
   i. 2 2 3
0  1  2
3  4  5

6  7  8
9 10 11
   <0;1;1
+-------+
|+-+-+-+|
||0|1|1||
|+-+-+-+|
+-------+
   (<0;1;1) { i. 2 2 3
4
```

If not all axes are specified, the selectors are applied starting with the leading axis and any axes left over at the end are taken in full:

```
   (<0;1) { i. 2 2 3
3 4 5
```

Each of the selectors may contain either a scalar or a list. If a selector contains a scalar, the corresponding axis will disappear from the shape of the result, as in the examples above. If a selector contains a list, even a list with only one item, the corresponding axis will remain in the shape of the result (its length will be the length of the selection list):

```
   (<0;1 0) { i. 2 2 3
3 4 5
0 1 2
```

We select two rows of item number 0. The rows stay in the order we requested them, and the result has rank 2.

```
   (<0 1;1 0;2) { i. 2 2 3
 5 2
11 8
```

Understand where each number came from. We are taking a 2×2 array of 1-cells, but only item 2 from each 1-cell. That leaves a 2×2 result.

```
   (<0;,1) { i. 2 2 3
3 4 5
   $ (<0;1) { i. 2 2 3
3
   $ (<0;,1) { i. 2 2 3
1 3
```

In the last example we are requesting a list of 1-cells; even though the list has only one item, its axis remains in the result.

If a selector contains a box (rather than the usual numeric), it calls for ***complementary selection*** along that axis: the contents of that box (i. e. the contents of the contents of the selector) indicate the indexes to *exclude* from the selection, and all other indexes are selected. Such a selector is considered to be specifying the list of non-excluded indexes, so the corresponding axis remains in the result. Example:

```
   (<0;1;<<1)
+---------+
|+-+-+---+|
||0|1|+-+||
|| | ||1|||
|| | |+-+||
|+-+-+---+|
+---------+
   (<0;1;<<1) { i. 2 2 3
3 5
```

We select a single 2-cell, and from that a single 1-cell, and within that we select all except item 1. The result is a 2-item list. Note that we had to put an extra **<** after the last **;** to ensure that the contents of the last selector was boxed.

```
   (<0;(<0$0);2)
+--------+
|+-+--+-+|
||0|++|2||
|| |||| ||
|| |++| ||
|+-+--+-+|
+--------+
   (<0;(<0$0);2) { i. 2 2 3
2 5
```

Complementary indexing can be used to select all of an axis, as in this example. We request all of axis 1 except the named items, and then we name an empty list: we get all the items. This trick is called for when we need to specify a selector for some axis after the axis we want to take completely (trailing axes can be taken in full simply by omitting their selectors).

If our use of `x { y` does not require any selector to specify a list (i. e. each selector is a scalar), we are allowed to omit the boxing of the selectors. This leaves `x` as a boxed numeric list (or scalar) in which the successive items indicate the single index to be selected from each axis. This form, in which `x` is `<i,j,k…`, corresponds to C's **array[i][j][k]…** .

```
   <0 1
+---+
|0 1|
+---+
   <0;1
+-----+
|+-+-+|
||0|1||
|+-+-+|
+-----+
   (<0 1) { i. 2 2 3
3 4 5
   (<0;1) { i. 2 2 3
3 4 5
```

The results are identical.

As a final simplification, if the selection is just a single item from axis 0, the left operand of dyad `{` may be left unboxed. This is the form in which we first met dyad `{` . Now that you have learned dyad `{` completely, take this quiz: what is the difference between `0 1 { y` and `(<<0 1) { y`?

```
   0 1 { i. 6
0 1
   (<<0 1) { i. 6
```

```
0 1
```

Answer: the results are identical, but because the left rank of dyad `{` is 0, `0 1 { y` applies dyad `{` twice, once for each atom of `0 1`, and collects the results into an array, while `(<<0 1) { y` applies dyad `{` just once.

Split String Into J Words: Monad `;:`

Monad `;:` splits a string of characters into J words, putting each word into a box of its own. Each word is a list of characters, so the result of monad `;:` is a list of boxed lists:

```
   ;: 'Words, words; and more words.'
+-----+-+-----+-+---+----+------+
|Words|,|words|;|and|more|words.|
+-----+-+-----+-+---+----+------+
```

Monad `;:` is a handy way to get boxed character strings, or to break an input stream into words if your language has word-formation rules similar to J's. Be aware that if the operand has an unmatched quote, monad `;:` will fail.

Fetch From Structure: Dyad `{::`

Dyad `{::` (this is a single primitive) has left rank 1, right rank infinite. It selects an atom from an array of boxes and opens it; it is therefore analogous to the `.` **(class member) operator** in C:

```
   1 {:: 'abc';1 2 3; i. 2 2
1 2 3
```

Item 1 was selected and opened.

`x {:: y` can go through multiple levels of structure referencing at once. If `x` is a list of boxes, the first box must be a valid left argument to dyad `{`; it is used to select an item of `y`, which is then opened; the next box of `x` selects an item from that opened box of `y`, which item is then opened; and so on till `x` is exhausted:

```
   struct1 =: 'abc' ; 1 2 3
   ]struct2 =: 'def';struct1;4 5 6
+---+-----------+-----+
|def|+---+-----+|4 5 6|
|   ||abc|1 2 3||     |
|   |+---+-----+|     |
+---+-----------+-----+
```

Here we have a structure in which item 1 is another structure.

```
   1 {:: struct2
+---+-----+
|abc|1 2 3|
+---+-----+
```

We select and open item 1, resulting in the enclosed structure.

```
   (1;1) {:: struct2
1 2 3
```

We select and open item 1 of **`struct2`**, and then open item 1 of the enclosed structure.

```
   (1;<<1 0) {:: struct2
+-----+---+
|1 2 3|abc|
+-----+---+
```

If the last selection specifies a list of items, as in this example, the selected boxes are not opened. **Note** that the Dictionary's description of dyad **`{::`** incorrectly indicates that the boxes are opened.

```
   (1;<<,1) {:: struct2
+-----+
|1 2 3|
+-----+
```

Even if the list contains only one item, it is not opened.

Only the last box of **`x`** may specify selection of a list of boxes:

```
   ]a =. <"0 i. 3 3
+-+-+-+
|0|1|2|
+-+-+-+
|3|4|5|
+-+-+-+
|6|7|8|
+-+-+-+
```

`a` is a 3×3 array of boxes.

```
   (1;1) {:: a
|rank error
|   (1;1)    {::a
```

The first selection took item 1 of **`a`** . This was a 3-item list of boxes, and it is inadmissible to open the list and perform further selections.

Note that if `x` is unboxed, dyad `{::` first boxes it and then uses it for selection. The Dictionary's description does not mention the boxing step.

Report Boxing Level: Monad `L.`

Monad `L.` has infinite rank and tells you the ***boxing level*** of its operand. Boxing level is defined recursively: if y is unboxed or empty, its boxing level is 0; otherwise its boxing level is one greater than the maximum of the boxing levels of its opened items:

```
   1 ;< 2 ;< 3 ;< 4
+-+---------+
|1|+-+-----+|
| ||2|+-+-+||
| || ||3|4|||
| || |+-+-+||
| |+-+-----+|
+-+---------+
   L. 1 ;< 2 ;< 3 ;< 4
3
   L. {. 1 ;< 2 ;< 3 ;< 4
1
```

You can use monad `L.` to decide how many levels of boxing to remove:

```
   >^:L. <<<6
6
```

Note that an empty boxed list shows boxing level of 0, but the type revealed by `3!:0` is 'boxed'. Also, fill elements for an empty boxed list are the boxed fill element a: :

```
   L. 0$a:
0
   3!:0 (0$a:)
32
   3 {. 0$a:
++++
||||
++++
```

---

17. Verb-Definition Revisited

Before we discuss verbs in more detail, let me point out that the colon character (`:`), which you have seen used mostly as a suffix to create new primitives, has its own meaning when it is used alone or as the first character of a primitive: `:` is a conjunction, and so are `:.` and `::` . The dot (`.`) also has a meaning but we won't be discussing it for a while. If you want `.` or `:` not to be treated as a suffix, make sure you precede it by a space: `3 : 'y.'` defines a verb, while `3: 'y.'` applies the verb `3:` to the noun `'y.'` producing the result `3` .

What really happens during `m :n` and `verb define`

Now that you're getting older it's time we had a little talk. About verbs. Up till now the verb-definition sequence has been exotic and peculiar, with a unique form; a possessor of great power but shrouded in mystery. We know only that it somehow breathes life into lines of text:

```
name =: verb define
verb definition
)
```

or

```
name =: verb : 'verb definition'
```

leaving ***name*** a verb.

With a layer of cosmetics scrubbed off it is more approachable. In both cases the conjunction `m :n` is at work. `m` must be a number, and `n` may be either a character string or the special number `0` (a boxed list is also possible but we consider it a curiosity). `m :n` executes the conjunction `:` with the arguments `m` and `n`; the result of this execution is an entity which can be any of the primary parts of speech (noun, adverb, conjunction, verb), and `m` indicates which one: `m`=0 means noun, 1 means adverb, 2 means conjunction, 3 means verb, 4 means dyadic verb; 10-13 have meanings we will learn later. You can remember the numbers 0-2 because nouns take no operands, adverbs 1, and conjunctions 2; verbs are last in the precedence order so they get number 3, and dyadic verbs with their extra operand get number 4. If **`n`** is a character string, it supplies the text of the entity (which, again, is given the part of speech indicated by **`m`**); if **`n`** is **`0`**, the interpreter interrupts what it's doing at the moment it executes the **`:`** conjunction, and reads lines from its input source until it finds one that contains only the word **`')'`**; execution of the interrupted sentence then continues, with the text of those lines becoming the right argument to **`:`** and thence the text of the defined noun/adverb/conjunction/verb. If the verb is defined in a script file, the input source (for lines to satisfy **`m :0`**) is the script file; otherwise the input source is the keyboard. If you forget the **`)`** in a definition at the end of a script file, the interpreter will switch over to keyboard input and you will find the system unresponsive until you satisfy it by typing **`)`** .

So what's with this '**`verb define`**' lingo? Simple: when you start J, you get a few names defined automatically, and **`verb`** and **`define`** are two of them—as you can see by asking the interpreter what their values are:

```
   verb
3
   define
:0
```

—so when you use **`verb define`** you are really executing **`3 :0`** to produce a verb; similarly **`dyad : '`*`one-line definition`*`'`** is the same as **`4 : '`*`one-line definition`*`'`** which executes the **`:`** conjunction to produce a dyadic-verb result.

The point is that a verb-definition sequence is just an instance of a compound verb produced by a conjunction, and the resulting verb can appear anywhere a verb is allowed. You may assign the verb to a name but that's not required. Here are some examples showing how a definition is just an ordinary part of a sentence:

```
   addrow =: monad : '+/y.' "1
```

We define the verb, using the **`:`** conjunction, and then we give the resulting verb a rank using the **`"`** conjunction. This is a principle to live by: **Always make sure any verb you define has the proper rank**. Following this rule will save you untold trouble by guaranteeing that the verb is applied to the rank of cell that you intended. The verb produced by the **`:`** conjunction has infinite rank; here, we expect to apply our verb on lists, so we assign a rank of 1 before we assign the verb to the name **`addrow`** .

```
   (3 : '+/y.') i. 6
```

```
15
```

See? we define a verb, then we execute it. It doesn't have to be assigned to a name. The distinction between code and data is not sharp as it is in C.

```
   a =. '+/';'*/'
   b =. (0{::a),'y.'
   (3 : b) 1 2 3 4 5 6
21
```

b is `'+/y.'` so we can use it as the text of a verb which we then execute.

```
   b =. (1{::a),'y.'
   (3 : b) 1 2 3 4 5 6
720
```

Here **b** is `'*/y.'` . In a later chapter we will explore the delights that result from being able to change the text of a program while you are executing it.

Remember: make sure the verbs you define have the proper rank.

Compound Verbs Can Be Assigned

Since we now understand that **verb define** is just a conjunction producing a verb result, and we know that we can assign its result to a name, we wonder whether we are allowed to assign any compound verb to a name. Yes indeed, we can, as we saw in the assignment to **addrow** in the previous section. Any verb can be assigned to a name:

```
   dotprod =: +/@:*"1
```

Dot-product of two lists is the sum of the products of respective pairs of elements in the lists.

```
   1 2 3 dotprod 1 2 3
14
```

dotprod takes the dot-product.

```
   veclength =: %:@:(dotprod~)"1
```

The length of a vector is the square root of its dot-product with itself.

```
   veclength 1 2 3
3.74166
```

Dual-Valence verbs: `u :v`

u :v also defines a verb of infinite rank but it is completely different from **m : n** . The defined verb is **u** if it is used monadically, but **v** if used dyadically:

```
   bv =. (monad : '+/y.') : (dyad : 'y. - x.')
   bv i. 7
21
   2 bv i. 7
_2 _1 0 1 2 3 4
```

You can see that the appropriate valence was chosen based on whether the verb was executed as a dyad or as a monad.

`u :v` is often used to assign a default left operand to a dyadic verb:

```
   pwr =: 2&pwr : (dyad : 'x.^y.')
```

If you execute `pwr` as a dyad you get `x^y` . If you execute it as a monad you get `2 pwr y` which is then executed (using the *dyadic* valence of `pwr`) to become `2 ^ y` :

```
   3 pwr 4
81
   pwr 4
16
```

The Suicide Verb `[:`

The verb `[:` fails if it is executed. Use it as one side of `u :v` to produce a verb that fails if it is used with the wrong valence. Example:

```
   i. 100 110 120 130
|out of memory
|       i.100 110 120 130
```

Oops, we thought we were looking those values up in a list. We're lucky that we just ran out of memory—sometimes using the wrong valence of a verb can have catastrophic consequences. To prevent it, use

```
   dyadi =: [: : i.
   dyadi 100 110 120 130
|valence error: dyadi
|       dyadi 100 110 120 130
   100 130 150 dyadi 100 110 120 130
0 3 3 1
```

`[:` also has a special meaning when it appears in a *fork*, which we will encounter later.

Multi-Line Comments Using `0 :0`

We know that all comments in J start with `NB.` and go only to the end of the line. There is no way to extend a comment to more than one line, but there is a way to put a series of non-executing lines into a script without having to have `NB.` in each of them. You simply define a noun using `0 :0` and put your comment inside the noun:

```
   0 :0
This is the first line of a comment.
The lines of the comment will become a noun; but,
since the noun is not assigned to anything, it simply
disappears.
)
```

Final Reminder

Remember: make sure the verbs you define have the proper rank.

<<   >>   Contents   Help

18. `u^:_1`, `u&.v`, and `u :.v`

The Obverse `u^:_1`

What would it mean to apply a verb a negative number of times? Applying a verb `_1` times should undo the effect of applying the verb once. In J, `u^:n` applies the verb `u` `n` times, and `u^:_1` is defined as the verb that undoes the effect of applying `u` once. `u^:_1` is called the ***obverse*** in J, and where possible it is the same as the mathematical inverse of `u` (or `x&u` in the dyadic case). Examples:

```
   >: 4
5
   >:^:_1 (5)
4
```

Monad `>:` adds 1; its obverse subtracts 1.

```
   *&2^:_1 (10)
5
```

Monad `*&2` multiplies by 2; its obverse divides by 2.

```
   2 *^:_1 (10)
5
```

In the dyadic case, the obverse of `x&u` is applied. The obverse of `2&*` is `%&2` .

Not all verbs have obverses; you can see the obverse of a verb `v`, if there is one, by typing `v b. _1` :

```
   +&2 b. _1
-&2
   (%&5)@:(+&1) b. _1
-&1@:(5&*)
   $ b. _1
|domain error
```

There is no obverse for monad `$` .

Some of J's obverses are very ingenious, and you can have a pleasant afternoon experimenting to find them. Most of them are listed in the Dictionary under the `^:`

conjunction.

Apply Under Transformation: `u&.v` and `u&.:v`

Using its ability to apply the obverse of a verb, J provides a feature of great elegance and power with the conjunction **`u&.v`** (monadic or dyadic). **`u&.v`** (all of whose ranks are the ranks of *monad* **`v`**) executes **`u`** after a temporary change in representation given by monad **`v`** . Formally, **`u&.v y`** is **`v^:_1@u@v y`**; informally, **`x u&.v y`** is **`v^:_1 (v x) u (v y)`** applied to each cell of **`x`** and **`y`**, with the results collected as usual (we will learn a formal notation for this later). The idea is that you change the operands using the transformation given by **`v`**; then do your work with **`u`**; then invert the transformation with **`v^:_1`** . Examples:

```
   (<1) +&.> 4;5;6
+-+-+-+
|5|6|7|
+-+-+-+
```

We add **`x`** to **`y`**; the transformation is that we remove the boxing before we add, and put it back after we finish. The verb **`+&.>`** has rank 0 (since monad **`>`** has rank 0), so here the scalar **`x`** is replicated to the shape of **`y`** before the unboxing occurs. **`fndisplay`** shows the details, where **`open``** means the inverse of **`open`** :

```
   defverbs 'plus"0 open"0'
   (<1) plus&.open 4;5
+-------------------------+-------------------------+
|open` (open 1) plus open 4|open` (open 1) plus open 5|
+-------------------------+-------------------------+


   <.&.(10&*) 4 4.43 4.89
4 4.4 4.8
```

`<. y` finds the largest integer not greater than y; by temporarily multiplying by 10 we truncate **`y`** to the next-lower tenth.

We can easily define a verb to take the arithmetic mean (i. e. the average) of a list:

```
   mean =: monad : '(+/ y.) % #y.'
   mean 1 2 4
2.33333
```

If we want to take the geometric mean, we could define a new verb to multiply and take the root, or we could just take the arithmetic mean of the logarithms and then undo the logarithm:

```
   mean&.(^."_) 1 2 4
2
```

Note that we had to use a verb of infinite rank as **`v`** so that **`u`** would be applied to the entire list. This is a common enough pattern that the conjunction **`&.:`** is provided which is just like **`&.`** but with infinite rank. We could have used **`mean&.:^.`** here.

To add 10 minutes to a time represented as hours,minutes,seconds, we can transform to seconds after midnight, do the addition, and transform back:

```
   0 10 0 +&.(24 60 60&#.) 13 55 0
14 5 0
```

Suppose we had a list of boxed scalar numbers, and we wanted to add them and leave the result boxed. How can we do it?

```
   ]a =. <"0 i. 6
+-+-+-+-+-+-+
|0|1|2|3|4|5|
+-+-+-+-+-+-+
```

The easy way is

```
   < +/ > a
+--+
|15|
+--+
```

but after you get used to **`&.`**, you will find that

```
   +/&.:> a
+--+
|15|
+--+
```

seems clearer, because it expresses the temporary nature of the unboxing/reboxing. As an exercise, take the time to see why

```
   +&.>/ a
```

gives the same answer but

```
   +/&.> a
```

does not.

Defined obverses: `u :.v`

`u :.v` has the same ranks as **`u`**, and it produces the same result as **`u`**, except that the obverse of **`u :. v`** is **`v`** . By defining verbs with appropriate obverses, you make it possible to use **`&.`** and **`^:_1`** with them. For example, in a finance application it is necessary to deal with dates in both (year,month,day) form and 'market day' form (for example, if Friday is market day number 1200, the following Monday will be market day number 1201). If you have written routines to convert between them:

```
ymdtomd =: dyad : 'definition'
mdtoymd =: dyad : 'definition'
```

you would be well advised to make them obverses of each other:

```
ymdtomd =: dyad : 'definition' :. mdtoymd
mdtoymd =: dyad : 'definition' :. ymdtomd
```

so that if you want to find the (y,m,d) date of the next market day after the date **`ymd`**, you simply code

```
1&+&.ymdtomd ymd
```

and J will convert **`ymd`** to a market day, add one, and convert back to (y,m,d) form.

u&:v and u&v

`x u&:v y` has infinite rank and is the same as **`(v x) u (v y)`** . It resembles **`u&.y`** but without the application of the obverse. It is just a way of saving a few keystrokes. **`x u&v y`** is like **`x u&:v y`** except that its ranks are both the same as the rank of monad **`v`** . As with **`@:`** and **`@`**, you are advised to stick to **`u&:v`** unless you are sure you need **`u&v`** .

The monadic forms **`u&v y`** and **`u&:v y`** are equivalent to **`u@v y`** and **`u@:v y`** respectively. I recommend that you use the **`@`** forms rather than the **`&`** forms, because your code will be full of **`m&v`** and **`u&n`** and it will reduce confusion if you don't have unnecessary **`u&v`** as well.

An observation about dyadic verbs

We noted earlier that usually if a dyad **`x v y`** is not symmetric (i. e. if **`x`** and **`y`** are treated differently), the **`x`** operand is the one which is more like control information and the **`y`** operand is the one more like data. We can see now that this is a consequence of the definition of **`x u^:v y`** : the verb that is applied repeatedly, or the verb whose obverse is taken, is **`x&u`**; only the value of **`y`** changes between executions of **`u`** . When you define dyadic verbs, you should take care to follow the same rule in assigning the left and right operands.

---

19. Performance: Measurement & Tips

J lets you express your ideas tersely, but it is up to you to make sure they are good ideas. Since each keystroke in a J sentence can summon up billions of machine cycles, you must make sure that you don't force the interpreter into a bad choice of algorithm. This will be hard at first, when you are struggling to figure out what the interpreter is doing, never mind how it is doing it; fortunately the interpreter gives you tools to measure the performance of your code.

Timing Individual Sentences

If you run the **JforC** script with

```
load 'system\packages\misc\jforc.ijs'
```

it will define the verb **Ts**. **Ts** stands for 'time and space' and it tells you how long it takes to run a given J sentence, and how much space the interpreter used during the execution. For example:

```
   a3 =. i. 1000
   Ts '+/\ a3'
4.3581e_5 5248
```

We define a noun **a3**, and we calculate the running total of its items. It took 0.00004 seconds to create the 1000-item running total, and used 5248 bytes. We could have done the whole operation in one line with **Ts '+/\ i. 1000'**, but the monad **i.** uses time and space too, so if we want to find out only what is used by **+/**, we make sure that's all we measure.

We can use **Ts** to start to understand what can make J programs slow. Let's define a verb to do the addition operation:

```
   sum =: dyad : 'x. + y.'"0
```

sum is an exact replacement for dyad **+**, having the same rank and function. Replacing **+** with **sum** does not change the result of a sentence:

```
   +/\ i. 7
0 1 3 6 10 15 21
   sum/\ i. 7
0 1 3 6 10 15 21
```

But the performance is quite different, as we can measure:

```
   a10 =. i. 10
   1000 Ts '+/\ a10'
2.68191e_5 1280
```

Because `+/\` is so fast, we give `Ts` a left argument to report the average time over 1000 runs. If we just ran the sentence once, the result would be corrupted by small timing variations introduced by the operating system. `sum/\` is not so fast so we run it only once:

```
   Ts 'sum/\ a10'
0.00181867 3648
```

Quite a difference: in this running total `sum` seems to be about 50 times slower than `+` . Let's just try adding a list to itself (remember that `u~ y` is equivalent to `y u y`):

```
   1000 Ts '+~ a10'
2.68191e_5 896
   100 Ts 'sum~ a10'
0.00033021 2560
```

Yes, `sum` is definitely slower than `+`, though only by a factor of 10 or so this time. Why should it be slower? The answer is, Because it deals with atoms. Since J verb-definitions are not compiled, but interpreted line-by-line on each execution, every single time we add two numbers with `sum`, the interpreter has to parse `'x. + y.'` and perform the addition. Why, it's a miracle that it only slows down by a factor of 10! The lesson is that if you define verbs with small rank, the interpretive overhead will be significant.

Still, that doesn't fully explain why `sum/\` is **so** much slower than `+/\` . Let's investigate further by increasing the size of the operand:

```
   a20 =. i. 20
   1000 Ts '+/\ a20'
2.68191e_5 1280
   Ts 'sum/\ a20'
0.00728641 3648
```

`+/\` is unchanged when we move to a list of 20 items—the operation is so fast that time is being spent starting the verb rather than running it—but `sum/\` slows down noticeably. Interesting; let's try bigger and bigger operands:

```
   a40 =. i. 40
   1000 Ts '+/\ a40'
2.76572e_5 1408
```

```
   Ts 'sum/\ a40'
0.0299561 4160

   a100 =. i. 100
   1000 Ts '+/\ a100'
2.76572e_5 1664
   Ts 'sum/\ a100'
0.185741 5184

   a400 =. i. 400
   1000 Ts '+/\ a400'
3.77143e_5 3200
   Ts 'sum/\ a400'
3.00367 11328
```

Holy cow! On a 400-item list, `sum/\` is 80000 times slower than `+/\`! What happened?

Recall what monad `sum/\` is really doing. It applies monad `sum/` to the first item of the list; then to the list made of the first 2 items; then the list made of the first 3 items; and so on. At each evaluation of monad `sum/`, the dyad `sum` verb is interleaved between the items and the result is evaluated *right-to-left*. The problem is, the interpreter doesn't analyze `sum` to know that it is associative—that `x sum (y sum z)` is the same as `(x sum y) sum z`—so it doesn't know that it can use the result from one subset as an input to the operation for the next subset, and it winds up performing every single addition: for the 400th item it adds all 400 numbers together. That's why its time increases as the square of the length of the list.

Monad `+/\` is fast because the interpreter knows that dyad `+` is associative, and therefore it reuses the result from one subset as input to the next, producing each item of the result with a single addition.

Well then, can we give a hint to the interpreter that `sum` is associative? Alas, no, but we have another trick up our sleeve. Consider monad `sum/\.`, which applies monad `sum/` to successive *suffixes*. If the interpreter is clever, it will notice that if it starts with the smallest suffix—the one made up of just the last item—and processes the suffixes in order of increasing size, it will always be evaluating `x sum (`***`previous suffix result`***`)`, and right-to-left evaluation implies that the result of the previous suffix can always be used as the right operand to each application of monad **`sum`**, without needing any knowledge of associativity. Let me tell you, this interpreter is nothing if not clever, and that's just what it does. All we have to do is to convert our **`sum/\`** into a variant of **`sum/\.`** . The way to do that is simple: we reverse the order of the items, apply **`sum/\.`**, and reverse the order again:

```
   sum/\.&.|. i. 7
0 1 3 6 10 15 21
```

This arises enough to be a standard J idiom: use it whenever you need to apply an associative verb on prefixes. It's much faster:

```
   Ts 'sum/\.&.|. a400'
0.014805 59264
```

Still not as fast as **`+/\`**, but the suffix version uses time proportional to the number of items rather than the square of the number of items.

Compounds Recognized by the Interpreter

The interpreter recognizes a great many compounds and has special code to perform the compound functions. For example, we have learned that **`u@:v y`** gives the same result as **`u v y`**, but it does not follow that the two forms are identical: **`+/ @:, y`** is faster than **`+/ , y`** . How do know what forms are handled with special code?

An appendix to the Dictionary gives a list of special code in the interpreter (press F1 to bring up help; then click on 'Dic' at the top of the page to bring up the Contents page; the appendices are listed at the end of the contents). There we see that there is special code for **`f/@:,`** so we know to use that form. Similarly, farther along we see that **`x i.&1@:< y`** has special coding, so we know to prefer that form over **`(x < y) i. 1`** . This list changes from release to release, so you should review it occasionally.

J's performance is very good even if you pay no attention whatsoever to which compounds have special coding, but if you're going to code a lot of J you might as well learn the interpreter's preferred idioms.

Use Large Verb-Ranks! and Integrated Rank Support

'Think big' is a watchword not just for program design, but for coding as well.

Starting a primitive has a small cost, but if you start a primitive for each atom of a large array, the cost will add up. To reduce the time spent starting primitives, apply them to the biggest operands possible. This means, **Use as large a verb-rank as you can**. See what a difference a tiny change can make:

```
   a =. i. 100000 10
   Ts 'a -@+ a'
3.96384 4.19552e6
   Ts 'a -@:+ a'
0.12801 8.3895e6
```

These two verbs produce identical results, but `-@+` is 30-fold slower than `-@:+` on this large operand. The reason is that `-@+` has rank 0 (taken from the rank of `+`), while `-@:+` has infinite rank. Rank 0 means that each pair of atoms is fed individually through the verb. So, when `-@+` is executed, two primitives are started for each pair of atoms, one to add and the other to change the sign. Execution of `-@:+` requires only two primitive-starts for the entire array.

You do not need to worry much about the ranks at which individual primitives are applied, because of an important feature of J called ***integrated rank support***. When a verb with integrated rank support is used as the `u` in `u"n`, the resulting verb runs with a single primitive-start and the application of the verb on the proper cells is handled within the primitive. So,

```
   100 Ts 'a + a'
0.0623949 4.19501e6
   100 Ts 'a +"0 a'
0.248846 4.19526e6
   100 Ts 'a +"1 a'
0.0681035 4.19526e6
   100 Ts 'a +"2 a'
0.0626361 4.1952e6
```

All these forms produce identical results. The weak dependence of the speed on the rank is typical of a verb with integrated rank support. Fastest execution is achieved when the verb is used alone, but the form `u"n` still runs fast, and the higher the rank, the less the loop-control overhead. The Special Code page referred to in the previous section includes the long list of the primitives with integrated rank support. You will see there that `u/`, `u/\`, and the like are also taken care of.

The practical effect of integrated rank support is that you don't need to worry much about using the largest possible rank for primitives. In compounds and verbs that you write, you do need to keep the rank high:

```
   Ts '(<a:;1) { a'
0.00939758 525568
   Ts '1 {"1 a'
0.00952329 525184
```

Integrated rank support in dyad **{** gives the two forms equal performance. Look what happens when we replace the **{** by a user-defined verb with the same function:

```
   from =. {
   Ts '(<a:;1) from a'
0.00953335 525760
   Ts '1 from"1 a'
0.365966 525696
```

from lacks integrated rank support, even though it is defined to have the same function as **{**, and it suffers when it is applied to each 1-cell. This is a good reason for you to learn the J primitives and not replace them with mnemonic equivalents.

Shining a Light: The J Performance Monitor

A magnet makes it easy to pick up a needle, but it won't much help you find a needle in a haystack. Likewise, being able to time and tune individual sentences will not suffice to let you improve the performance of a large J program. A large program spends most of its time executing a small subset of its code, and any improvements you make to other areas are simply wasted effort. I remember a case where a 20,000-line assembler-language program was spending 30% of its time executing a single machine instruction—and that instruction turned out to be unnecessary! What you need is a tool that will direct your attention to the areas where a speedup will really matter.

The J Performance Monitor will show you how much time is spent executing each line of your application. You can run the Lab on the Performance Monitor to see all the facilities available, or you can jump right into timing your code with the simple sequence

```
   load 'jpm'
```

Do this once to load the tool. Then, for each timing run, execute

```
   start_jpm_ 1e7
357142
```

The operand of **start_jpm_** is the size in bytes of the trace buffer, and the result is the number of trace entries that can fit in the buffer. A trace entry is added for each line executed, and for entry and exit of explicit definitions (i. e. verbs defined with **`verb define`**).

```
   run the code you want to time
   viewtotal_jpm_ ''
```

J will display a popup window with information about the time spent in each verb. An example display is

```
+---------+-----+--------+--------+-----+----+---+
|name     |locale|all     |here    |here%|cum%|rep|
+---------+-----+--------+--------+-----+----+---+
|accpay   |base  |0.001435|0.000829| 57.8| 58 |1  |
|intrep   |base  |0.000213|0.000213| 14.8| 73 |1  |
|accint   |base  |0.000393|0.000147| 10.2| 83 |1  |
|stretch  |base  |0.000142|0.000142|  9.9| 93 |1  |
|intexpand|base  |0.000105|0.000105|  7.3|100 |1  |
|[total]  |      |        |0.001435|100.0|100 |   |
+---------+-----+--------+--------+-----+----+---+
```

The columns contain the following information:

name the name of the verb

locale the locale the verb was running in (we will discuss locales in a later chapter)

all the amount of time spent in this verb including time spent in verbs called by this verb

here the amount of time spent in this verb but **not** including time spent in verbs called by this verb

here% the **here** time as a percentage of total time

cum% cumulative total of **here%**

rep the number of times the verb was executed

You should focus your attention on the **here** column. If you see a verb that is taking longer than you think it should, double-click on its name to look at the details of its execution. Double-clicking on **`accpay`** will pop up another window showing

```
+--------+--------+---
+----------------------------------+
|all     |here    |rep|
accpay                            |
+--------+--------+---
+-----------------------------------+
|0.000041|0.000041|1  |
monad                             |
|0.000040|0.000040|1  |[8] if. 4~:#y.
do.                |
|0.000000|0.000000|0  |[9] 'imm frq int pay' return.
end.|
|0.000054|0.000054|1  |[10] 'm f i p'=.
y.                |
|0.000116|0.000116|1  |[11] len=.$p=.f#p%
f               |
|0.000724|0.000131|1  |[12] j=.}.len accint f intrep
i   |
|0.000322|0.000322|1  |[13] r=.j*+/\p%m}.1,(m-1)}.
j      |
|0.000137|0.000137|1  |[14] (len$(-f){.1)
#r              |
|0.001435|0.000841|1  |total
monad                       |
+--------+--------+---
+-----------------------------------+
```

We see that line 13 takes the most time. Clicking on the column heading will sort the lines using that column as a key, making it easy for you to concentrate on the individual lines that are taking the most time.

The J Performance Monitor makes it easy to give your code a good finish by pounding down the nails that are sticking up. As of Release 5.01a there are a few quirks you need to work around: you cannot have a verb with the same name as a locale; you must close a detail window before you create a new one; and time spent in explicit modifiers is not correctly accounted for.

---

20. Input And Output

Foreigns

In J, all file operations are handled by ***foreigns*** which are created by the ***foreign conjunction*** **`!:`** . The foreigns are a grab-bag of functions and you would do well to spend some time glancing over their descriptions in the Dictionary so you'll have an idea what is available. All the foreigns defined to date in J are of the form **`m!:n`** with numeric **`m`** and **`n`**, and they are organized with **`m`** identifying the class of the foreign. For example, the foreigns for operations on files have **`m`**=1.

File Operations `1!:n`; Error Handling

The foreigns **`1!:n`** perform file operations. For ease of use I have given several of them names in **`jforc.ijs`** . To see the details of what they do, read the Dictionary.

Monad **`1!:1`** (**`ReadFile`**) has rank **`0`** . **`1!:1 y`** takes a filename as **`y`** and produces a result that is a list of characters containing the text of the file. In our examples, the filename is a boxed character string; **`y`** can be a file number but we won't get into that.

```
   s =. 1!:1 <'system\packages\misc\jforc.ijs'
   s
NB. File definitions for'J For C Programmers'
NB. Copyright (c) 2002 Henry H. Rich
…
```

Dyad **`1!:2`** (**`WriteFile`**) has rank **`_ 0`** . **`1!:2 y`** writes to the file **`y`**, using the character string **`x`** to provide the contents. Any existing file **`y`** is overwritten. **`y`** is a boxed character string or a file number.

```
   ('Test Data',CR,LF) 1!:2 <'c:\Temp\temp.dat'
```

Dyad **`1!:3`** (**`AppendFile`**) has rank **`_ 0`** . **`x 1!:3 y`** appends the character string **`x`** to the of file **`y`** (creating the file if it does not exist):

```
   ('Line 2',CR,LF) 1!:3 <'c:\Temp\temp.dat'
```

Monad **`1!:55`** (**`EraseFile`**) has rank **`0`** . **`1!:55 y`** erases the file **`y`** with no prompting. Be careful: the power of J makes it possible for you to delete every file on your system with one sentence.

The special file number **`2`** sends output to the screen. The verb monad **`Display`** in **`jforc.ijs`** uses file number **`2`** to display its operand (which must be a character string) on the screen; you can put **`Display`** sentences in a long verb to see intermediate results. In most cases you will prefer to use **`printf`**, described below.

There are many other **`1!:n`** foreigns to manage file locks, query directories, handle index read/write, and do other useful things. The Dictionary describes them, and I will add only that the description of **`1!:12`** is misleading: the length to be written is implied by the string argument **`x`** and must not be included in **`y`**; therefore item 1 of **`y`** is a boxed scalar, rather than a boxed list of 2 integers as for **`1!:11`** .

Error Handling: `u ::v`, `13!:11`, and `9!:8`

When you deal with files you have to expect errors, which will look something like

```
   s =. 1!:1 <'c:\xxx\yyy.dat'
|file name error
|   s=.    1!:1<'c:\xxx\yyy.dat'
```

indicating that the file was not found.

You can set up your verbs to catch errors instead of interrupting with a message at the console. We will learn one way here and another later when we study control structures for J verbs. The compound **`u ::v`** has infinite rank, and can be applied either monadically or dyadically. **`u ::v`** executes the verb **`u`** first (monad or dyad, as appropriate); if **`u`** completes without error, its result becomes the result of **`u ::v`**; but if **`u`** encounters an error, **`v`** is then executed, and its result becomes the result of **`u ::v`** :

```
   rerr =: 1!:1 :: (13!:11@(''"_))
```

`rerr y` will execute **`1!:1`** to read **`y`**; if that fails it will execute the foreign **`13!:11 ''`** . **`13!:11 ''`** produces as result the error number of the last error encountered. This means that if **`1!:1`** succeeds, the result of **`rerr`** will be a string, while if **`1!:1`** fails, the result will be a number:

```
   rerr <'system\packages\misc\jforc.ijs'
NB. File definitions for'J For C Programmers'
NB. Copyright (c) 2002 Henry H. Rich
…
```

```
   rerr <'c:\xxx\yyy.dat'
25
```

You could use **`3!:0`** to see whether the result of **`rerr`** is a string or a number (**`2`** means string, **`1`**, **`4`**, **`8`**, or **`16`** means number). If you want to see the error message associated with an error number, there's a foreign **`9!:8`** to give you the list of errors:

```
   25 { 9!:8 ''
+-----------------+
|file number error|
+-----------------+
```

Treating a File as a Noun: Mapped Files

Rather than reading a file and assigning the data to a noun, you can leave the data in a file and create a noun that points to the data. The data will be read only when the noun is referred to. This is called ***mapping*** the file to a noun.

J's facilities for mapping files are described in the Lab 'Mapped Files'. A quick example of a mapped file is

```
   require 'jmf'
   JCHAR map_jmf_ 'text'; 'system\packages\misc\jforc.
ijs'
```

after which the noun **`text`** contains the data in the file:

```
   45 {. text
NB. File definitions for'J For C Programmers'
```

Moreover, if you assign a new value to **`text`**, the file will be modified.

If you are dealing with large files, especially read-only files or files that don't change much, mapping the files can give a huge performance improvement because you don't have to read the whole file. You must be very careful, though, if you map files to nouns, because there are unexpected side effects. If we executed

```
   temp =: text
   temp =: 'abcdefgh'
```

we would find that the value of **`text`** had changed too! (If you try this, use a file you don't mind losing). The assignment of **`text`** to **`temp`** did not create a copy of data of **`text`**, and when **`temp`** was modified, the change was applied to the shared data. If a file is mapped to a noun, you have to make sure that the noun, or any copy made of the noun in any verb you pass the noun to, is not changed unless you are prepared to have some or all of the other copies changed. This topic is examined in greater depth under 'Aliasing of Variables' in the chapter on DLLs.

If the faster execution is enticement enough for you to take that trouble, you can consult the Lab to get all the details.

Format Data For Printing: Monad And Dyad `":`

Since J reads and writes all files as character strings, you will need to be able to convert numbers to their character representation. Dyad `":` has rank `1 _` . `y` may be numeric or boxed, of any rank (we will not consider boxed `y` here). If `x` is a scalar it is replicated to the length of a 1-cell of `y` . If `y` is numeric, each 1-cell of `y` produces a character-list result, with each atom in the 1-cell of `y` being converted to a character string as described by the corresponding atom of `x` and with the results of adjacent 0-cells being run together; the results from all 1-cells are collected into an array using the frame with respect to 1-cells. That sounds like the description of a verb with rank 1; almost so but not quite, because `":` looks at the entire `y` and adds extra spaces as necessary to make all the columns line up neatly.

A field descriptor in `x` needs two numbers, ***w*** and ***d*** . These are represented as the real and imaginary parts of a complex number, so that only a single scalar is needed to hold the field descriptor. The real part of the field descriptor is ***w***, and it gives the width of the field in characters. If the result will not fit into the field, the entire field is filled with asterisks; to prevent this you may use a ***w*** of 0 which will cause the interpreter to make the field as wide as necessary. The imaginary part of the field descriptor is ***d***, giving the number of digits following the decimal point. If either ***w*** or ***d*** is less than zero, the result is in exponential form (and the absolute value of ***w*** or ***d*** gives the corresponding width), otherwise the result is in standard form.
Examples:

```
   0 ": i. 4 4
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
```

Note that extra spaces were added to the one-digit values to make them line up. Note also that 'enough space' includes a leading space to keep adjacent results from running together.

```
   1 ": i. 4 4
```

```
0123
4567
89**
****
```

When you specify a width, you are expected to mean it, and no extra space is added. Here two-digit results would not fit and were replaced with `'*'`.

```
   0 0j3 0j_3 ": 100 %~ i. 3 3
0 0.010  2.000e_2
0 0.040  5.000e_2
0 0.070  8.000e_2
```

A complex number has its parts separated by `'j'` . Here the first field is an integer, the second has 3 decimal places, and the third is in exponential form.

When dyad `":` is applied to an array, the result has the same rank as `y` . If you need to write that result to a file, you will need to use monad , to convert it to a list, possibly after adding `CR` or `LF` characters at the ends of lines.

Monad `":`

Monad `":` resembles dyad `":` with a default `x`, except that `x` depends on the value of the corresponding atom of `y` . The simple description of monad `":` is that it formats numbers the way you'd like to see them formatted. The full description is as follows: The precision ***d*** is given by a system variable that can be set by the foreign `9!:11 y` (and queried by `9!:10 y`; initially it is `6`) ; as a quick override you may use the fit conjunction `":!.`***d*** to specify the value of ***d*** for a single use of `":` . If an atom of `y` is an integer, a field descriptor of `0` is applied; if the atom has significance in the first 4 decimal places, a field descriptor of `0j`***d*** is applied; otherwise a field descriptor of `0j_3` is applied. Trailing zeros below the decimal point are omitted.

```
   9!:10 ''
6
   ": 0 0.01 0.001 0.000015 0.12345678 2
0 0.01 0.001 1.5e_5 0.123457 2
```

In spite of all the detail I gave about the default formatting, in practice you just apply monad `":` to any numeric operand and you get a good result. Monad `":` accepts character arguments as well and leaves them unchanged:

```
   ; ":&.> 'Today ';'is ';2002 1 24
Today is 2002 1 24
```

We went inside the boxes with **`&.>`** and formatted each box's contents with monad **`":`**; this made all the contents strings and we could run them together using monad **`;`** .

Monad **`":`** also converts boxed operands to the character arrays, including boxing characters, that we have seen in the display of boxed nouns.

Format binary data: `3!:n`

If you need to write numbers to files in binary format, you need something that will coerce your numbers into character strings without changing the binary representation. J provides the foreigns **`3!:4`** and **`3!:5`** for this purpose. Each is a family of conversions, invoked as a dyad where the left operand selects the conversion to be performed.

`2 (3!:4) y` converts the numeric list **`y`** to a character string, representing each item of **`y`** by 4 characters whose values come from the binary values of the low-order 32 bits of **`y`** :

```
   16b31424334
826426164
```

This is an integer whose value is **`0x31424344`**. We can convert it to a character string:

```
   2 (3!:4) 826426164
4CB1
```

The 4 characters (with values **`0x34`**, **`0x43`**, **`0x42`**, **`0x31`**) correspond to the bits of the number **`826426164`** in little-endian order.

We can use **`a. i.`** to look at the codes for each character in case they are not printable:

```
   a. i. 2 (3!:4) 1000 100000
232 3 0 0 160 134 1 0
```

`232 3 0 0` corresponds to **`1000`** and **`160 134 1 0`** to **`100000`** .

`_2 (3!:4) y` is the inverse of **`2 (3!:4) y`**, converting a character string to integers:

```
   _2 (3!:4) '4CB15CB1'
826426164 826426165
```

The other integer conversions are similar. **`1 (3!:4) y`** converts the low-order 16 bits of each item of **`y`** to 2 characters, and **`_1 (3!:4) y`** converts back to integers, 2 characters per integer. **`0 (3!:4) y`** is like **`_1 (3!:4) y`** but the integers are unsigned (i. e. in the range **`0-65535`**).

The floating-point conversions are analogous. **`2 (3!:5) y`** converts each item of **`y`** to 8 characters representing long floating-point form, and **`_2 (3!:5)`** y converts back; **`1 (3!:5) y`** and **`_1 (3!:5) y`** use 4-character short floating-point form.

`printf`, `sprintf`, and `qprintf`

When you need formatted lines for printing you may feel at home using **`printf`** and **`sprintf`**, which work like their C counterparts. **`printf`** displays a line, while **`sprintf`** produces a string result:

```
   'The population of %s is %d\n' printf
'Raleigh';240000
The population of Raleigh is 240000

   s =. 'The total of %j is %d.\n' sprintf 1 2 3;+/1 2 3
   s
The total of 1 2 3 is 6.
```

You need to execute

```
load 'printf'
```

to get the **`printf`** verbs defined. J's **`printf`** contains a few features beyond C's such as the **`%j`** field type seen above. You should run the Lab 'Formatting with printf' for details.

One feature with no analogue in C is **`qprintf`**, which produces handy typeout for debugging:

```
   a =. 3 4 5 [ b =. 'abc';'def';5 [ c =: i. 3 3
   qprintf 'a b c '
a=3 4 5 b=
+---+---+-+
|abc|def|5|
+---+---+-+
 c=
0 1 2
3 4 5
6 7 8
```

`qprintf` is described in the 'Formatting with printf' lab.

Convert Character To Numeric: Dyad `".`

Dyad `".` has infinite rank. `x ". y` looks for numbers in the words of `y`, which must be a character array. Words representing valid numbers are converted to numbers; words not representing valid numbers are replaced by `x` . If 1-cells of `y` have differing numbers of numbers, `x` is used to pad short rows. 'Valid numbers' to dyad `".` are anything you could type into J as a number, and more: the negative sign may be `'-'` rather than `'_'`; numbers may contain commas, which are ignored; and a number may start with a decimal point rather than requiring a digit before the decimal point. With the relaxed rules you can import numbers from a spreadsheet into J, using `x&".` to convert them to numbers:

```
   999 ". '12 .5 3.6 -4 1,000 x25'
12 0.5 3.6 _4 1000 999
```

All legal numbers except for `'x25'` .

<<   >>   Contents   Help

21. Calling a DLL Under Windows

Interfacing to a DLL is one thing you can do the old-fashioned way: by getting a copy of a working program and editing it. You can find starter programs in the J distribution. A good one to start with is **`\system\packages\winapi\registry.ijs`**. You can glance through this program to see what it does. It starts with

```
require 'dll'
```

which defines the verb **`cd`** . Calls to the DLL appear in lines like

```
rc =. 'Advapi32 RegCreateKeyExA i   i *c i *c i i i *i
*i'
      cd   root;key;0;'';0;sam;0;(,_1);(,_1)
```

(this is a single line in the program; I have shown it here as 2 lines because it won't fit on the page as a single line)

This example exhibits the elements of a call to a DLL. The left operand of **`cd`** is a character list describing the function to be called and what arguments it is expecting. Here we are calling the entry point **RegCreateKeyExA** in the library **Advapi32**. The sequence of **`i`**s and **`*c`**s describes the interface to the function. The first item in that sequence describes the type of value returned by the function; the other items are the arguments to the function and are a one-for-one rendering of the argument list that would be passed in C. So, the line above is appropriate for calling a function defined with the prototype

int RegCreateKeyExA(int, char *, int , char *, int, int, int, int *, int *);

The descriptors can be **`c`** (**char**), **`s`** (**short**) , **`i`** (**int**), **`f`** (**float**), **`d`** (**double**), **`j`** (complex), or **`n`** (placeholder—a value of **`0`** is used and the result is ignored); all but **`n`** can be preceded by **`*`** to indicate an array of that type. **`*`** by itself indicates an array of unspecified type.

The right operand of **`cd`** is the actual arguments to be passed to the called function. It must be a list of boxes in which the contents of each box holds one argument to the called function (there is no box to correspond to the returned value). An argument whose description does not include **`*`** must correspond to a boxed scalar of the appropriate type. An argument described with **`*`** must correspond to a box whose contents are either an array or a boxed memory address (see below). The address of the array or memory area is passed to the DLL and the DLL may modify the array or memory area; this is how you return an array from the function.

`s` and `f` are not native J types. A scalar argument described by `s` or `f` is converted by the interpreter to the correct form, while an argument described by `*s` or `*f` is assumed to point to a memory area with the correct format.

`i` and `d` descriptors pose a bit of a problem. They both correspond to numbers in J, but unfortunately it is up to you to ensure that the J internal representation of the number matches the type expected by the DLL. There is no officially-sanctioned way to do this, but as of J5.02 you can use monad `<.` to ensure that a value is an integer (for `i`) and monad `_&<.` to ensure that a value is floating-point (for `d`).

Note that if you get things wrong and the function scribbles outside the bounds of an array, J may crash. Note also that a vector of `0`s and `1`s is held inside J as a vector of Booleans, which are **char**s. When the function calls for a vector of **int**s, J will convert any Boolean to **int**. In the example above, `(,_1)` reserved space for an **int**; `(,0)` would have worked too but would require a conversion.

When the function returns, its returned value will be boxed and appended to the front of the boxed list that was the right operand of `cd` to produce the result of the execution of `cd` . You may use this returned value as you see fit. Any box that contained an array may have had its contents modified by the function; you may open the box to get the changed value.

If J was unable to call the DLL, the `cd` verb fails with a domain error. You can then execute the sentence `cder ''` which will return a 2-element list indicating what went wrong. The User Guide gives a complete list of errors; the most likely ones are `4 0` (the number of arguments did not match the number of declarations), `5 x` (declaration number `x` was invalid—the count starts with the declaration of the returned value which is number 0), and `6 x` (argument number `x` did not match its declaration—the count starts with the first argument which is number 0 and must match the *second* declaration).

Memory Management

Passing arrays into the called function is adequate only for simple functions. If the function expects an argument to be a structure, possibly containing pointers to other structures, you will have to allocate memory for the structures, fill the structures appropriately, and free the memory when it is no longer needed. J provides a set of verbs to support such memory management.

Allocate memory: mema

`mema` ***`length`*** allocates a memory area of ***`length`*** bytes. The result is the address of the memory area, as an integer. It is **`0`** if the allocation failed. **`mema`** has infinite rank.

You must box the memory address before using it as an operand to **`cd`** . Do not box the address for use as an operand to **`memf`**, **`memw`**, or **`memr`** .

Free memory: memf

`memf` ***`address`*** frees the memory area pointed to by ***`address`*** . ***`address`*** must be a value that was returned by **`mema`** . Result of **`0`** means success, **`1`** means failure. **`memf`** has infinite rank.

Write Into a Memory Area: memw

`data` **`memw`** ***`address,byteoffset,count,type`***
causes ***`data`*** to be written, starting at an offset of ***`byteoffset`*** from the area pointed to by ***`address`***, for a length of ***`count`*** items whose type is given by ***`type`*** . ***`type`*** is **`2`** for characters, **`4`** for integers, **`8`** for floating-point numbers, **`16`** for complex numbers; if omitted, the default is **`2`** . If ***`type`*** is **`2`**, ***`count`*** may be one more than the length of ***`data`*** to cause a string-terminating NUL (**`\0`**) to be written after the ***`data`*** .

Read From a Memory Area: memr

`memr` ***`address,byteoffset,count,type`***
produces as its result the information starting at an offset of ***`byteoffset`*** from the area pointed to by ***`address`***, for a length of ***`count`*** items whose type is given by ***`type`*** . ***`type`*** is **`2`** for characters, **`4`** for integers, **`8`** for floating-point numbers, **`16`** for complex numbers; if omitted, the default is **`2`** . The result is a list with ***`count`*** items of the type given by ***`type`*** .If ***`type`*** is **`2`**, ***`count`*** may be **`_1`** which causes the read to be terminated before the first NUL (**`\0`**) character encountered.

Aliasing of Variables

When a noun is assigned the value of another noun, as in

```
   a =. b =. 5
```

a single memory area is used to hold the value common to both nouns, and the two nouns are said to be ***aliases*** of each other. Aliasing obviously reduces the time and space used by a computation. The interpreter takes care to ensure that aliasing is invisible to the programmer; if after the statement above we execute

```
   b =. 6
```

the interpreter will assign the new value to **`b`** only, leaving **`a`** unchanged. What actually happens is that the new value is created in a data block of its own and the descriptor for the noun **`b`** is changed to point to the new block. (Almost all verbs create their outputs in newly-allocated data blocks. As of release 5.03 the exceptions are **`]`**, **`[`**, and **`,`** and **`u}`** when used in one of the forms that produces in-place modification. Increasing the number of cases recognized for in-place execution is a continuing activity of the J developers).

If there were nothing more to say about aliasing, I would not single it out for mention from among the dozens of performance-improving tricks used by the interpreter. What makes it worth considering is the effect aliasing has when elements outside the J language touch J's nouns. This can occur in two ways: when a noun is mapped to a file and when a noun is modified by a DLL.

Aliasing of Mapped Nouns

When a noun is mapped to a file, the descriptor for the noun points to the file's data and that pointer is *never changed* even if a value is assigned to the variable: the whole point of mapping the noun to the file is to cause changes in the noun to be reflected in the file, so any assignment to the noun causes the data to be copied into the area that is mapped to the file.

In addition, when **`b`** is a noun mapped to a file and is assigned to another noun as with

```
   a =. b
```

the noun **`a`**, which is aliased to **`b`**, also inherits the 'mapped-to-file' attribute. This behavior is necessary to make mapped files useful, because assignments to **`x.`** and **`y.`** are implicit whenever a verb is invoked and it would defeat the whole purpose of mapping if the data of the file had to be copied every time the mapped noun was passed to a verb. The combination of aliasing and mapping means that any assignment to a mapped noun also changes the values of all other nouns which share the same mapping: for example, if you pass a mapped noun **`a`** as the right operand of a verb that modifies its **`y.`**, **`y.`**, **`a`**, and the data in the file will all be modified.

Keeping track of the aliasing is the price you pay for using mapped files. If you need to copy a noun making sure you get a fresh, unmapped data block, you must not assign the mapped noun directly, but instead assign the result of some verb that creates its output in a new data block. For example, as of release 5.01 the assignment

```
   a =. 1&# b
```

will create a new data block containing the data of **`b`**, and **`a`** will point to that new block.

Aliasing of DLL Operands

The J interpreter uses aliasing for boxed cells of an array, so that if you execute

```
   b =. i. 10000 10000
   a =. b;5
```

item 0 of **`a`** simply contains a pointer to the data block of **`b`** rather than a fresh copy of the 400MB array. In addition, when a list of boxes is used as the right operand of **`cd`**, as in

```
   'dll-spec' cd   root;key;0;'';0;sam;0;(,_1);(,_1)
```

any array operands to the DLL function are passed via a pointer to the data in the boxed list, with no separate copy of the data being made. This means that if the DLL modifies one of its arguments, any nouns aliased to that argument will also be modified: if the DLL function called above modifies its argument 1, the noun **`key`** and any noun aliased to **`key`** (possibly including private nouns in suspended verbs) will be changed. To protect yourself from such side-effects, you can use **`(1#key)`** in place of **`key`** in the invocation of **`cd`** .

---

22. Socket Programming

J provides the standard set of socket functions. This chapter assumes you already understand socket programming and provides a description of J's facilities.

First, you must load the script that defines the socket interface:

```
   require 'socket'
```

(**`require`** is like **`load`** but if the file has already been **`load`**ed it does nothing). This will define a number of variables inside J. You must make the variables visible to your program, and the easiest way to do that is with the verb **`DefSockets`** defined in **`system\packages\misc\jforc.ijs`**:

```
   DefSockets ''
```

You are then ready to create a socket:

```
   sdsocket AF_INET,SOCK_STREAM,0
+-+--+
|0|96|
+-+--+
```

`sdsocket` returns ***`return_code;socket_number`*** . Nonzero return codes indicate errors; look in ***`J_directory/`*****`system/main/socket.ijs`** for definitions. The valid address families and socket types are defined in ***`J_directory/`*****`system/main/defs/netdefs_`*****`YourOS`*****`.ijs`**.

sdselect

You check the status of sockets with

```
   sdselect read_list;write_list;error_list;maxtime
```

The three lists are lists of sockets and ***`maxtime`*** is a time in milliseconds. **`sdselect`** will delay for a maximum of ***`maxtime`*** milliseconds until it sees a socket in ***`read_list`*** that is ready for reading, or one in ***`write_list`*** that is ready for writing, or one in ***`error_list`*** that has an error. When it finds a qualifying socket, or when the time limit is reached, it returns ***`result_code;read_ready_list;write_ready_list;error_list`*** which lists all the qualifying sockets. A ***`maxtime`*** of 0 always returns immediately, giving the current status of the specified sockets.

If **`sdselect`** is invoked with an empty operand (**`sdselect ''`**), it checks all sockets with a ***`maxtime`*** of 0.

Asynchronous Sockets and `socket_handler`

By default, sockets created by **`sdsocket`** are ***blocking*** sockets; an operation on such a socket waits until data is available. This can tie up your J session while the remote computer is responding, so if you are serious about your socket programming you will want to make your sockets ***nonblocking***. You do so with

```
   sdasync socket_number
```

which marks that socket as nonblocking and requests notification of changes in its status. All operations to a nonblocking socket return immediately (with the error code **`EWOULDBLOCK`** if the operation cannot be immediately completed). When there is a change in the status of a nonblocking socket, the interpreter executes

```
   socket_handler ''
```

and it is up to you to have a **`socket_handler`** defined that will use **`sdselect`** to see what sockets are ready for transfers and then execute those transfers.

Names and IP Addresses

Sockets are addressed by IP address and port number. To translate a domain name to an IP address, use

```
   sdgethostbyname domain_name
```

for example,

```
   sdgethostbyname 'www.jsoftware.com'
+-+-+---------------+
|0|2|216.122.139.159|
+-+-+---------------+
```

where the result is ***`return_code;address_family;address`*** . If the domain is unknown an address of **`'255.255.255.255'`** is returned.

```
   sdgethostbyaddr address_family;address
```

will translate an IP address back to a domain name.

The name of your machine is given by

```
   sdgethostname ''
+-+-----+
|0|1qfe3|
+-+-----+
```

(result is ***return_code;name***) so you can get your own IP address by

```
   sdgethostbyname 1 {:: sdgethostname ''
+-+-+-----------+
|0|2|65.80.203.8|
+-+-+-----------+
```

If you have a socket with an active connection you can get some information about the machine on the other end with

```
   sdgetpeername socket_number
+-+-+------------+--+
|0|2|64.58.76.229|80|
+-+-+------------+--+
```

where the result is ***return_code;address_family;remote_IP_addr;*** ***remote_port*** .

You can get information about your own end of a connected socket with

```
   sdgetsockname socket_number
+-+-+-----------+----+
|0|2|65.80.203.8|2982|
+-+-+-----------+----+
```

where the result is ***return_code;address_family;local_IP_addr;*** ***local_port*** .

Connecting

Reading from a Web site, you use the sequence **sdsocket**/**sdconnect**/**sdsend**/**sdrecv**/**sdclose** .

```
   sdconnect socket;address_family;IP_addr;port
```

will connect your socket to the remote machine addressed by ***address_family;*** ***IP_addr;port*** . If the socket is blocking, **sdconnect** completes when the connection has been made. If the socket is nonblocking, **sdconnect** will return immediately with the error code **EWOULDBLOCK** and **socket_handler** will be called when the connection has been made. An example is

```
   sdconnect 184;2;'64.58.76.229';80
```

With the connection established, you can send data with

```
   data sdsend socket;flags
```

where the ***flags*** are any of the **MSG_** values from **socket.ijs**, usually 0. The result is ***`return_code;number_of_bytes_sent`*** . Fewer bytes may be sent than were in your data; you will have to resend the excess.

You receive data with

    sdrecv socket,count,flags

where ***`count`*** is the maximum number of bytes you will accept and ***`flags`*** are any of the **`MSG_`** values from **`socket.ijs`**, usually 0. The result is ***`return_code;data`*** . If the socket is blocking, **`sdrecv`** will wait until it is ready for reading; if the socket is nonblocking, **`sdrecv`** will immediately return (presumably you issued the **`sdrecv`** only after using **`sdselect`** to verify that the socket was ready for reading). In either case, **`sdrecv`** will return with ***`data`*** if it has any; if the length of ***`data`*** is 0, that means that the connection was closed by the peer and all data sent by the peer has been received.

When you have finished all you data transfers for a socket, ,you must close it with

    sdclose socket

which has as result an unboxed ***`return_code`*** .

Listening

If you want to wait for a remote machine to get in touch with you, use the **`sdbind`**/**`sdlisten`**/**`sdaccept`** sequence instead of **`sdconnect`** . The sequence is:

    sdbind socket;address_family;IP_addr;port

to establish a connection between the ***`socket`*** and the address given by ***`address_family;IP_addr;port`*** . If ***`IP_addr`*** is **`''`**, the socket can be used to listen for a connection to any address on the machine. If ***`port`*** is 0, the system will assign a port number. The result of **`sdbind`** is an unboxed ***`return_code`*** .

    sdlisten socket;connection_limit

causes the operating system to start listening for connections to the address bound to ***`socket`*** . The result is an unboxed ***`return_code`*** .A maximum of ***`connection_limit`*** connections can be queued awaiting **`sdaccept`** . When a connection is made to a listening socket's address, the socket is shown in **`sdselect`** as ready to read, and you should issue

    sdaccept socket

which will return ***`return_code;clone_socket`*** where ***`clone_socket`*** is a new socket, with all the attributes of ***socket*** but additionally with a connection to the remote host. You should direct all your **sdsend**, **sdrecv**, and **sdclose** operations to the ***clone_socket*** .

Other Socket Verbs

Datagrams

The sequences given above apply to sockets of type **SOCK_STREAM**, e. g. TCP sockets. Connectionless sockets with type **SOCK_DGRAM** are also supported, and could be used for UDP transfers. The verbs to use to transfer datagrams are

sdrecvfrom *socket;count;flags*

where ***count*** is the maximum number of bytes you will accept and ***flags*** are any of the **MSG_** values from **socket.ijs**, usually 0. The result is ***return_code; data;sending_address*** . If the socket is blocking, **sdrecvfrom** will wait until it is ready for reading; if the socket is nonblocking, **sdrecvfrom** will immediately return (presumably you issued the **sdrecvfrom** only after using **sdselect** to verify that the socket was ready for reading). The returned ***data*** is the datagram, which may have a length of 0.

data* sdsendto *socket;flags;address_family;IP_addr; port

where the ***flags*** are any of the **MSG_** values from **socket.ijs**, usually 0. The datagram is sent to the remote address given by ***address_family;IP_addr; port***, and the result is ***return_code;number_of_bytes_sent*** .

Socket Options

Verbs to query and set socket options are

sdgetsockopt *socket;option_level;option_name*

with result ***return_code;option_value*** . Examples:

```
sdgetsockopt sk;SOL_SOCKET;SO_DEBUG
sdgetsockopt sk;SOL_SOCKET;SO_LINGER
```

sdsetsockopt *socket,option_level,option_name, value_list*

(note that the operand is an **unboxed** list) The option is set, and the result is ***return_code;option_value*** . Examples:

```
sdsetsockopt sk,SOL_SOCKET,SO_DEBUG,1
sdsetsockopt sk,SOL_SOCKET,SO_LINGER,1 66
```

`sdioctl` ***`socket,option,value`***

reads or sets control information (result is ***`return_code;value`***). Examples:

```
   sdioctl sk,FIONBIO,0  NB. set blocking
   sdioctl sk,FIONBIO,1  NB. set non-blocking
   sdioctl sk,FIONREAD,0  NB. count ready data
```

Housekeeping

`sdgetsockets  ''`

The result is ***`return_code;list_of_all_active_socket_numbers`*** .

`sdcleanup ''`

All sockets are closed and the socket system is reinitialized. The result is **`0`** .

---

23. Loopless Code V—Partitions

The adverb \ operated on subsets of the operand **y** that were taken in a regular way. Now we will take the next step, and operate on irregular subsets of **y** . This will finally give us the chance to do interesting work with character strings.

Find Unique Items: Monad `~.` and Monad `~:`

Monad **`~.`** has infinite rank. **`~. y`** is **y** with duplicate items removed and is called the ***nub*** of **y**. The items of **`~. y`** are in the order of their first appearance in **y** :

```
   ~. 'Green grow the lilacs'
Gren gowthliacs
   ]a =. _2 ]\ 0 1 1 2 0 1 2 3 1 3 1 2
0 1
1 2
0 1
2 3
1 3
1 2
   ~.a
0 1
1 2
2 3
1 3
```

y can have any rank, and **`~. y`** gives the unique items.

Monad **`~:`** has infinite rank and tells you which items monad **`~.`** would pick. **`~: y`** is a Boolean list where each element is **1** if the corresponding item of **y** would have been selected for the nub of **y** :

```
   ~: 'Green grow the lilacs'
1 1 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 1
   ~: a
1 1 0 1 1 0
   (~: a) # a
```

```
0 1
1 2
2 3
1 3
```

`~. y` is equivalent to `(~: y) # y` .

Apply On Subsets: Dyad `u/.`

Dyad `u/.` has infinite rank. `x u/. y` applies `u` to subsets of `y` for which the corresponding items of `x` (called the ***keys***) are identical. I will skip the formal description in favor of a verbal one: First find the nub of `x`, call it ***`nx`*** . Then, for each item of ***`nx`***, find all matching items of `x`; make an array of the corresponding items of `y` (this array will always have rank one more than the rank of an item of `y`), and apply `u` to that array; the result becomes one item of the result of `x u/. y` . The items of `x u/. y` thus correspond to items of the nub of `x` . Note that the subsets of `y` may have different shapes (they will all have the same rank, being made of items of `y`, but each subset may have a different number of items). For this reason, we usually have `u` produce a boxed result to avoid framing fills.

```
   ]a =. 2 0 1 0 2 </. 'Hello'
+--+--+-+
|Ho|el|l|
+--+--+-+
```

The subsets were created, and each one was boxed. Note that each subset is a list, even the one with only one element:

```
   $&.> a
+-+-+-+
|2|2|1|
+-+-+-+

   3 0 3 0 3 +//. 100 1 200 2 300
600 3
```

The summing was performed for each subset. Note that the result is sorted not on the value of the keys, but rather on the smallest index of each key in `x` .

```
   ]a =. _2 ]\ 'Fred';100;'Joe';200;'Fred';50;'Sam';30
+----+---+
|Fred|100|
+----+---+
|Joe |200|
+----+---+
|Fred|50 |
+----+---+
|Sam |30 |
+----+---+
```

A small database of amounts we owe. We can quickly get a total for each creditor:

```
   ]b =. ({."1 a) +/@:>/. ({:"1 a)
150 200 30
```

({."1 a) gives the list of names, which we use as keys for the data given by **({:"1 a)** . For each subset, we open the boxes and add up the results. Note that **+/@>/.** would not do in place of **+/@:>/.**, and understand why.

It would be nice to have names associated with the totals:

```
   (~. {."1 a) ,. <"0 b
+----+---+
|Fred|150|
+----+---+
|Joe |200|
+----+---+
|Sam |30 |
+----+---+
```

We box the totals using **<"0** before we join items of the totals to the items of the nub. Recall that **x ,. y** concatenates each item of **x** with the corresponding item of **y** .We take advantage of the fact that the results of **u/.** have the same order as the nub.

Apply On Partitions: Monad `u;.1` and `u;.2`

Unlike dyad **u/.** which applies **u** to scattered subsets of **y**, **u;.n** applies **u** to sequential subsets of **y** . In monad **u;.n**, the subsets are computed from information in **y** itself; in dyad **u;.n**, **x** specifies the subsets. **u;.n** is really 4 different cases distinguished by the number **n**; we will start with the case where **n** is **1**, **2**, **_1**, or **_2** .

Avoid the error of thinking that an operation on a subset of **y** somehow modifies **y** . Never happens. In J, the result of a verb is always a new noun. You may assign that noun to the same name as one of the operands, but until you do, the old operand is unchanged, and both it and the result of the verb are available for further use.

`u;.1 y` ***partitions*** `y` by finding the items of `y` that match the first item of `y` (i. e. `0 {y`); each such item, called a ***fret***, is the start of an ***interval*** of `y` which runs from the fret to the last item before the next fret (the last interval runs to the end of `y`). Monad `u` is applied to each interval (which is always a list of items of `y`), and the results of `u` become the items of the overall result of `u;.1 y` .

```
   <;.1 ' a list of words '
+--+-----+---+------+-+
| a| list| of| words| |
+--+-----+---+------+-+
```

Each `' '` character, even the one at the end, started a new interval. We are not restricted to boxing the intervals:

```
   #;.1 ' a list of words '
2 5 3 6 1
```

Here we report the length of each word.

`u;._1 y` is like `u;.1 y` except that the fret itself is omitted from the interval. The interval could be empty in that case:

```
   <;._1 ' a list of words '
+-+----+--+-----++
|a|list|of|words||
+-+----+--+-----++
   #;._1 ' a list of words '
1 4 2 5 0
```

`u;.2 y` and `u;._2 y` are like `u;.1 y` and `u;._1 y` except that the frets are those items of `y` that match its *last* item, and each marks the *end* of an interval:

```
   <;.2 'Mississippi'
+--+---+---+---+
|Mi|ssi|ssi|ppi|
+--+---+---+---+
```

Use `u;.2 y` to split a list when you know the ending marker for the intervals:

```
   ;: ;._2 (0 : 0)
Fred 500
Joe 200
Fred 50
```

```
Sam 30
)
+----+---+
|Fred|500|
+----+---+
|Joe |200|
+----+---+
|Fred|50 |
+----+---+
|Sam |30 |
+----+---+
```

Here `(0 : 0)` creates a noun from the lines that follow. Each line ends with an unseen `LF` character which provides a convenient fret for splitting the lines. We use `;._2` to apply monad `;:` to the text of each line (not including the fret, which is dropped by `;._2`). Monad `;:` splits each string into words and boxes the words. The result is not the same as the array we created earlier, because the second column is character rather than numeric. We could have written `noun define` instead of `(0 : 0)` .

Monad `u;.n` has rank `1 _` when `n` is `1`, `2`, `_1`, or `_2` .

Apply On Specified Partitions: Dyad `u;.1` and `u;.2`

When `n` is `1`, `2`, `_1`, or `_2`, `x u;.n y` has infinite rank and, in the one-dimensional case where `x` is a Boolean list of `0`s and `1`s, resembles `u;.n y` . The difference is that the frets are given by the positions of `1`s in `x` rather than by values of the items of `y` . We can define `u;.1 y` as `(({.y)="_ _1 y) u;.1 y`, and `u;.n` for the other values of `n` similarly.

```
   0 1 0 1 0 +/;.1 (20 30 40 50 60)
70 110
```

As in this example, some leading items of `y` may not be in any interval.

In the general case `x` is a list of boxed Boolean lists, with `j{::x` supplying the frets for axis `j` . The partition is then multidimensional:

```
   (0 1 1;1 0 0 1) <;.1 i. 3 4
+------+--+
|4 5 6 |7 |
+------+--+
|8 9 10|11|
+------+--+
```

In both the monadic and dyadic cases of `u;.1`, `u;._1`, `u;.2`, and `u;._2`, the partitions to which `u` is applied have the same rank as `y` (but the shapes along the leading axes are reduced by the partitioning).

Find Sequence Of Items: Dyad `E.`

Dyad `E.` has infinite rank and is used to slide a pattern across an array and look for positions at which the pattern matches the items of the array. `x` and `y` should have the same rank, and the sliding occurs along all axes of `y`, giving a Boolean result with the same shape as `y` . We will consider only the cases where `y` is a list.

```
   'is' E. 'Mississippi'
0 1 0 0 1 0 0 0 0 0 0
```

The `1`s in the result tell where the pattern starts. The result of `E.` is often used as input to `u;.1` :

```
   ('is' E. 'Mississippi') <;.1 'Mississippi'
+---+-------+
|iss|issippi|
+---+-------+
```

A more ambitious example:

```
   html =. 0 : 0
<th><a href='page1.html'>Press here to go back</a></th>
<th><a href='page2.html'>Press here to go home</a></th>
<th><a href='page3.html'>Press here to go away</a></th>
</table>
</center>
)
   ('<a' E. html) {.@:(<@:(8&}.);._1)@:('>'&,);.1 html
+------------+------------+------------+
|'page1.html'|'page2.html'|'page3.html'|
+------------+------------+------------+
```

That looks like a useful result, but what a mess to produce it! If you want, you may try to understand that, but a better approach is to break it up:

```
   extracthref =: <@:(8&}.) ;._1 @:('>'&,)
```

This seems understandable, with effort: the execution order is `((<@:(8&}.)) ;.`

`_1) @:('>'&,)`; we prepend `'>'` to `y`, then we use `;._1` on the string. That will make the prepended `'>'` the fret, and so we will break the string into parts that start with `'>'`, throw away the first 8 characters of each part, and box each trimmed-down part. We can even try it on a test bit:

```
   extracthref 'abcdefghijkl>xxx'
+----++
|ijkl||
+----++
```

Now we can look again at our original line, rewritten:

```
   ('<a' E. html) {.@:extracthref ;.1 html
+------------+------------+------------+
|'page1.html'|'page2.html'|'page3.html'|
+------------+------------+------------+
```

This makes some sense now. The execution order is `('<a' E. html) ({.@: extracthref ;.1) html` . We use `E.` to find all the starting positions of `'<a'` tags; then, for each one, we split what follows the `'<a'` into blocks terminated by `'>'`, and then we take the first one, which will have the data before the `'>'` that matched the `'<a'` .

Multidimensional Partitions

The left argument to the partitioning dyads `u;.n` can be a list of boxes. In this case the first box contains the partition marks for axis 0, then next box contains the marks for axis 1, and so on. If a set of partition marks is an empty list, the corresponding axis will be unpartitioned.

Apply On Subarray: Dyad `u;.0`

Dyad `u;.0` has rank `2 _` . `x u;.0 y` uses `x` to specify a subarray of `y`; the subarray is extracted, and `u` is applied to it to produce the final result. We will discuss the simple case where `x` is a rank-2 array.

The first item of `x` (call that *`s`*) gives the starting corner of the subarray. The second item of `x` gives the length of each axis of the subarray. If `x` is shorter than the rank of `y`, unspecified axes are taken in full. For example:

```
   ]a =. a. {~ (a. i. 'a') + i. 4 4
abcd
efgh
ijkl
```

```
mnop
```

A cute little expression in its own right. See why this produces the 4×4 array of characters shown. Remember that **`a.`** is the alphabet of all ASCII characters in order. An even more elegant way to produce the same result would be **`(i. 4 4)&+&.(a.&i.) 'a'`** .

```
   (0 0 ,: 2 2) ];.0 a
ab
ef
```

Starting corner **`0 0`**; lengths **`2 2`**; result is a 2×2 subarray, left unchanged by monad **`]`** .

```
   (1 2 ,: 3 2) ];.0 a
gh
kl
op
```

Starting corner **`1 2`**; lengths **`3 2`**; result is a 3×2 subarray, left unchanged by monad **`]`** . You get the idea.

If an item of ***`s`*** is negative, the corresponding axis of the subarray extends backward from the starting point (and its absolute value is used as the starting position):

```
   (2 _2 ,: 2 2) ];.0 a
jk
no
```

Starting corner **`2 2`** (the character **`'k'`**); lengths **`2 2`**, but running backward in axis 1; result is a 2×2 subarray, left unchanged by monad **`]`** . Note that the axis extends backward, but the items retain their normal order—you have merely specified the interval by its endpoint rather than its beginning point. If you want to reverse the order of the axis, you can do that too, by making the corresponding *length* negative:

```
   (2 _2 ,: 2 _2) ];.0 a
kj
on
```

Starting corner **`2 2`** (the character **`'k'`**); lengths **`2 2`**, but running backward in axis 1; result is a 2×2 subarray with axis 1 reversed, left unchanged by monad **`]`** .

The subarray must end at the limits of the array. If the length requests more than that, the subarray will be shorter than was requested. A length of _ or __ will always fetch from the starting point to the limit of the array.

Dyad **`u;.0`** is a great way to pick out a subarray to work on. Always consider using it whenever you find yourself using more than one application of **`{.`**and **`}.`** to select a portion of an array. Even if you are working with a list, using dyad **`u;.0`** is a good way to work on a portion of the list without copying any part you aren't working on.

Apply On All Subarrays: Dyad `u;.3` and `u;._3`

`x u;._3 y` applies **`u`** to subarrays of **`y`** as specified by **`x`** . The operation is similar to dyad **`u/`** (infix), but dyad **`u/`** slides a one-dimensional window across **`y`** to select sequences of items of **`y`**, while dyad **`u;._3`** moves a multidimensional window throughout **`y`** to select multidimensional regions. When the window has only one dimension, the operation is like dyad **`u/`** but with a little extra control over the positions of the window.

The second item of **`x`** (actually, **`| 1{x`**) gives the size of the window (if an item is negative the corresponding axis is reversed before **`u`** is applied to the window; an infinite value means 'take all the way to the end of the array'). The first item of **`x`** (**`0 {x`**) is the ***movement vector***: the window is positioned at every point at which each item of the window position is an integral multiple of the corresponding item of the movement vector, as long as the window fits inside **`y`** . If an item of the movement vector is smaller than the corresponding item of the size, the windows will overlap along that axis.

Dyad **`u;._3`** is useful in imaging applications. The following example averages 2×2-pixel regions in an image using a simple box filter:

```
   ]image =. ? 8 8 $ 100
31 88 65 15 68 38 38 49
14 58 84 59 95 55 14 98
40 14 56 25 48 46 96 12
19 31 62 12 65 62 80 24
47 38 20  2 90 42 14 94
41 13 88  9 16  7 36 25
13 78 45 34 45 80 93 65
21 67 90 25 86 47 50 60
   (2 2,:2 2) (*&0.25)@:(+/)@:, ;._3 image
47.75 55.75    64 49.75
   26 38.75 55.25    53
```

```
34.75 29.75 38.75 42.25
44.75  48.5  64.5    67
```

The rank of each operand presented to `u` is the same as the rank of `y` . The shape of each operand presented to `u` is the shape of `y` with the leading items replaced by `|1{x` (formally, this is `(|1{x),(#x)}.$y)`.

If `x` has rank less than 2, it is processed as if it were `1,:x` which will try the window at all possible locations.

`x u;.3 y` is similar to `x u;._3 y`, but the window is positioned at every starting point that is within `y`, even if the entire window will not fit within `y` . For those positions at which the window will not fit, the operand to `u` is truncated.

Extracting Variable-Length Fields Using `^:` and `;.1`

Breaking a string of variable-length fields into its individual fields seems to be a problem requiring a loop. How can you find the second field until you have examined the length of the first? There happens to be a good way to do this in loopless J. We will use as an example a set of character-string records where each record is preceded by a one-digit length (maximum string length is 9 characters). For example, the string

```
   data =. '5There2is1a4tide2in3the7affairs2of3men'
```

contains 9 records, each containing a single English word. How do we split the string into its records?

First, we examine each possible starting position and calculate how long a record would be if it started at that position. The calculated length must be the entire record length including that of the length field itself. Obviously, we will be calculating spurious lengths at the places that turn out not to be record-start points, but that is the small price we pay for loopless coding. Here, the data length is given by the difference between the ASCII value of each byte and the ASCII value of `'0'`, and we add one to account for the length of the length field:

```
   ]l =. >: (a. i. data) - a. i. '0'
6 37 57 54 67 54 3 58 68 2 50 5 69 58 53 54 3 ...
```

Next, we calculate, for each position, where the next record will start assuming that a record starts at that position. Clearly, we do this by adding the putative record length to the offset of the position. If the resulting total is past the end of the string, we limit the position to one character past the end of the string:

```
   ]n =. (#l) <. l + i. # l
6 38 38 38 38 38 9 38 38 11 ...
```

Now for the trick. The first record starts at offset 0. The next record starts at offset `0{n` . The record after that starts at offset `(0{n){n`, and so on. To get the list of all the starting positions, we use

```
   ]pos =. (n,_1) {~^:a: 0
0 6 9 11 16 19 23 31 34 38 _1
```

The special power `u^:a:` means 'keep applying `u` until the result stops changing, and return the vector of results'. It is like `u^:_` in that it applies `u` until the result stops changing, but `u^:a:` returns all the intermediate results from `u`, not just the last one. The result from `{~^:a:` is the list of start-of-record positions, and all that remains is to collect the records. We discard record pointers that are off the end, and box the records starting at thepositions given. We do this by converting the list of valid record-start points to a Boolean vector, and using the partitioning functions to box the records:

```
   ((i. #data) e. pos) <;._1 data
+-----+--+-+----+--+---+-------+--+---+
|There|is|a|tide|in|the|affairs|of|men|
+-----+--+-+----+--+---+-------+--+---+
```

---

24. When Programs Are Data

One characteristic of maturity in programming is readiness to pass a program as an argument. A well-designed program does not exhibit a megalomanic urge to do everything the user may desire; it is content to perform a limited function well, and to leave other functions to other programs. A suite of such programs can be variously connected to perform a great variety of functions, with each program doing its bit and passing control to the next one.

In C a program is passed to another program by pointer reference, and invoked by **(*pfi)(arguments)**. J has no pointers, but it has a great many ways to pass executable nuggets around the system. We will learn them now.

Calling a Published Name

The simplest way to get a verb `f` to pass control to another program `g` is to define `f` to call a verb with a public name, say `f_subfn`, and then to define `f_subfn` to be `g` . In C, this would be ridiculous, because it would imply that `f_subfn` is permanently bound to a single value of `g`, with the result that all calls to `f` from anywhere in your program would be stuck with the same value of `g` .

J overcomes this objection by allowing redefinition of `f_subfn` . Each invocation of `f` looks like

```
f_subfn =: g
f arguments
```

When `f` is invoked elsewhere, the correct value of `f_subfn` will be assigned similarly, so each invocation of `f` can pass control properly.

I find this technique hideous, but I have to admit it is effective. J uses it to get callbacks from DLLs. It must not be used if `f_subfn` is going to be called in response to an event, since its value may have been redefined by the time the event occurs.

Using the Argument To a Modifier

If the verb isn't going to call a name of its own choosing, you have to tell it what to call. The simplest way to do this is to change the verb into a modifier; then when it executes it has access to its operands, which can be verbs. So, instead of

```
f =: verb : 'definition'
```

you write

```
f =: adverb define
monadic definition
:
dyadic definition
)
```

and within the definition of `f` you can refer to the left operand of the adverb, which goes by the name `u.` . If you decide to write a conjunction instead, its right operand is `v.` . The noun operands of the derived verb (`u f` or `u f v`) are `x.` and `y.`, as usual. We will discuss user-defined modifiers in a later chapter.

In Chapter 3 this technique was used to calculate the Chebyshev coefficients of a function. The function to be approximated is one of the inputs to this calculation, so we write an adverb and let its left operand be the function. With `chebft` defined as in the example, an example of its use is:

```
   10 (2&o.) chebft 0 1
1.64717 _0.232299 _0.0537151 0.00245824 0.000282119…
```

Here the function to be approximated is the cosine function `2&o.`, evaluated for `10` Chebyshev coefficients over the interval `0 1` .

When you define a modifier, you have no way to specify that you are defining only the dyadic case as you can for a verb with `dyad define` . Instead, you use the form given above, in which the monadic definition (if any) is separated from the dyadic by a line containing a single `':'` character. `chebft` did this, since its derived verb is always dyadic.

Invoking a Gerund: `m`:6`

Sometimes you want to use a noun rather than a verb to designate a verb to be called. An asynchronous socket handler is an example: the socket handler will have many transfers going on at once, each one with a callback to be executed when the transfer is complete. The callbacks must be put into a table along with other information about the transfer; in other words, the callbacks must be nouns.

We have already met nouns that carried verbs; we called them gerunds. We found that gerunds were created by the conjunction ` and executed by `m@.v` . While these tools are adequate to allow verbs to be passed as arguments, some simplifications are available that we will discuss now.

A gerund created by ``u`v`` is always a list (each element of which, as we learned, is an atomic representation of a verb which we will treat as untouchable). Even if there is only one verb, the result of `` ` `` is a list:

```
   +`''
+-+
|+|
+-+
   $ +`''
1
```

It makes sense for the gerund created by `` ` `` to be a list, since it contains a list of verb-representations one of which is selected for execution by `m@.v`. But when we are passing a single verb-as-noun as an argument, it is OK for it to be a scalar box. And, it can be invoked using the ``` `: ``` conjunction: ``m`:6`` converts the gerund `m` into a verb. So, in our example, we pass the callback as one box in a parameter list, and then we select it, turn it into a verb with ``m`:6``, and execute it, as we can see in a stripped-down example. The gerund operations are not difficult so I am going to keep your interest with a couple of new tricks:

```
   callback =: dyad : '(x.) =: y.'
```

Howzat? `(x.) =:`? Yeah, this means that the *value* of `x.` tells what variable will be assigned. `x.` can be any valid assignment target: a name, a multiple assignment, or other exotic forms given in the Dictionary.

```
   calledfn =: monad : '(0 { y.)`:6 (1 {:: y.)'
```

So `calledfn` is expecting an argument of (at least) 2 boxes. The first one will be the gerund to execute, and the second one will be the argument to pass to that verb. We open the second box (with `{::`) but we leave the first as a box so that ``` `:6 ``` can turn it into a verb.

```
   calledfn 'vbl'&callback ` (<25)
```

The first challenge is figuring out what the argument to `calledfn` is. `` ` `` sees a verb on its left; it converts that to a gerund. It sees a noun to its right, so it appends it unchanged to the gerund from the left. This produces

```
   'vbl'&callback ` (<25)
+----------------------+--+
|+-+------------------+|25|
||&|+-------+--------+||  |
```

```
|| ||+-+---+|callback|||  |
|| |||0|vbl||        |||  |
|| ||+-+---+|        |||  |
|| |+-------+--------+||  |
|+-+------------------+|  |
+---------------------+--+
```

The first box is the atomic representation of the verb, just as mysterious as it was billed to be, and the second box has the 25. Now what do we get when we pass that in as the argument to `calledfn` (try to work it out before you peek at the answer)?

```
   calledfn 'vbl'&callback ` (<25)
25
```

Did you get it? We executed `'vbl'&callback 25` which then executed `vbl =: 25` which has the result `25`, which comes back as the result of `calledfn` . The assignment was public:

```
   vbl
25
```

Passing the Definition Of a Verb: `128!:2` (Apply)

As an alternative to passing a gerund and invoking it with `m`:6`, you could pass the string representation of a verb and make a verb out of it with `3 :n` or `4 :n` . Better yet, if you can make do with a monadic verb, you can use the foreign dyad `128!:2` which has rank `1 _` and goes by the name Apply. `x 128!:2 y` takes the string `x` which must describe a verb, and applies the verb so described to `y` (as a monad). It is therefore similar to `(3 : 'x') y` with the restriction that `x` must describe a one-line verb without assignments. The advantage of using `128!:2` rather than the method in the next section is that `y` does not have to be converted to its string representation.

Passing an Executable Sentence: Monad `".` and `5!:5`

As the ultimate in flexibility, you can pass an entire J sentence as a character string and then execute it with monad `".` . It is executed exactly as if it had been a line of the executing verb (or from the keyboard if that's where `".` was entered). For example:

```
   ". 'a =. i. 4'
```

```
0 1 2 3
```

The sentence was executed.

```
   a
0 1 2 3
```

The assignment was performed.

```
   ". '+/' , ": a
6
```

The operand of monad `".` must be a string, so before we can take its total we must convert `a` to a sequence of characters that will have the value of `a` when *executed*. For a large operand, converting to string form adds overhead that might steer you towards using a gerund or `128!:2` instead.

I think C programmers are likely to overlook opportunities to use monad `".` because it is so foreign to their experience. It is equivalent to compiling and executing a code segment as part of the running program—it's just unthinkable. But in J, it's commonplace.

If you are going to use monad `".` you will face the problem of converting your nouns to string form. Here's the display of a noun: what character string would you execute to produce a noun with that value?

```
   a
+------------+----------------------------------+
|+---+--+---+|0 2.25 4.5 6.75 9 11.25 13.5 15.75|
||abc|de|fgh||                                  |
|+---+--+---+|                                  |
+------------+----------------------------------+
```

Fortunately, you don't have to worry about it. The foreign monad `5!:5` takes as `y` a boxed *name* (not a value) and produces a string which when executed has the same value as the variable named by `y` . So:

```
   5!:5 <'a'
(<<;._1 ' abc de fgh'),<2.25*i.8
```

…and if you tell me you came up with the same string, I'm not going to believe you.

25. Loopless Code VI

The algorithms that are hardest to put into loopless form are the ones that chug through the items of an array, modifying a set of temporary variables that control the operation at each step. I am going to show you how to put one example of such an algorithm into loopless form. The resulting structure can be used for many such problems.

The example problem is a simulation. In a certain church most of the worshippers are devout, but the front pew is always packed with knaves. Collection is taken every Sunday. Each knave brings two coins to throw in, but, being a knave, he first removes the two most-valuable coins from the plate before adding his own. Given **p**, the contents of the plate when it reaches the front row (a list of coin-values) and **k**, the coins brought by the knaves (an n×2 array), what is the final contents of the plate, and what does each knave make off with? For test data, we will use

```
   p =. 100 25 100 50 5 10
```

giving the contents of the plate as it enter the knaves' row, ,and

```
   ]k =. _2 ]\ 50 50 100 25 5 10 25 50 25 10
 50 50
100 25
  5 10
 25 50
 25 10
```

as the coins to be thrown in by the greedy knaves.

After trying for a clever solution for a little while we give up and decide we are going to have to simulate the action of each knave. We start by writing a verb **knave** to perform a knave's action. The design of this verb requires a little careful joinery to make it useful later: we will invoke it as **x knave y** where **x** is an item of **k**, i. e. the input for one knave, and **y** is the result from applying **knave** for the previous knave; the result of **knave** must have the same format as the **y** operand of **knave**; finally, the result of **knave** must include whatever we want as the final solution of the problem.

The trick is that the result of **knave**, which will be the input to the next invocation of **knave**, must carry all the information needed by the next invocation of **knave**; this is the way information is passed from knave to knave. The main design decision is to figure out what the format of the **y** operand of **knave** will be.

Obviously we need to know the contents of the plate as it is given to each knave. Also, the purpose of **knave** is to calculate what coins are removed, so those coins should be part of the result. We decide that the **y** operand of **knave** will consist of those two things, in the order **(coins removed),(plate contents)**, and we already know that the **x** operand will have the format of an item of **k**, i. e. the knave's two coins. Now we are ready to code **knave** .

It should look at the plate-contents portion of its right argument, sort it into order of size, take the two biggest values as the result of interest, and use the rest (with the knave's own coins added) as the plate contents to be passed to the next knave. The coins that were removed are put into their place at the beginning of the result vector. In J this will be:

```
   knave =: dyad define
xlen =. #x.  NB. number of coins added/removed
splate =. \:~ xlen }. y.  NB. extract plate contents,
sort
(xlen {. splate) , (xlen }. splate) , x.  NB. build
result
)
```

Let's test it. The **y** operand to the first invocation of knave will have a couple of placeholders in the place where the coins removed by the previous knave would be, followed by the initial contents of the plate. In other words,

```
   ]inity =. ({:k),p
25 10 100 25 100 50 5 10
```

Here we used the last item of **k** as a placeholder. The values don't matter but we want the right shape so the program will still work if we change the number of coins in **k** . Applying this value to the first knave we get

```
   50 50 knave inity
100 100 50 25 10 5 50 50
```

Yes, that's right: the first two items of the result are the two dollar coins the knave took, and he threw his coins in at the end.

Before we go on we can't help noticing that taking the first two items of **splate** and then dropping those same items—that's needless work. We can simplify

knave to

```
   knave =: dyad : '(\:~ (#x.) }. y.) , x.'
```

Now we need to apply **knave** sequentially on all items of **k** . We have learned enough J to write a sentence to do that, but because this is a recurring problem I have written a conjunction **LoopWithInitial** to hide the complexity (we'll look at its details in a moment). This conjunction takes the verb **knave**, the initial value **inity**, and the array **k** and applies **knave** repeatedly, with each item of **k** taking a turn as **x** with the **y** set to the result from the previous invocation of **knave** :

```
   ]res =. knave LoopWithInitial inity   k
100 100 50 25 10 5  50 50
 50  50 50 25 10 5 100 25
100  50 25 25 10 5   5 10
 25  25 10 10  5 5  25 50
 50  25 10 10  5 5  25 10
```

We see the result after each application of **knave** (if you want to see the input alongside the result, type `k ,&<"_1 res`). The contents of the plate are included in **res** as well; we can extract the desired result simply as

```
   2 {."1 res
100 100
 50  50
100  50
 25  25
 50  25
```

Once you have defined the verb and the initial value, you can use **LoopWithInitial** to solve problems of this kind.

You may skip the rest of this chapter if you are not curious about how **LoopWithInitial** works. It performs the following steps:

```
   LoopWithInitial =: conjunction define
by =. <"_1 y.     NB. 1 box items of y
ry =. |. by       NB. 2 reverse order for \.
ey =. ry , <n.    NB. 3 append initial value
r  =. u.&.>/\. ey NB. 4 apply u. in boxes
er =. }: r        NB. 5 remove initial value
rr =. |. er       NB. 6 put in original order
>rr               NB. 7 remove boxing
)
```

Since the initial value is going to be appended to the `y` operand, and they have dissimilar shapes, it is necessary to box each item of `y` as well as the initial value. Once the items of `y` are boxed, they are put into reverse order, because as we have seen `u/\.` is much faster than `u/\` . Then the initial value is appended to the reversed boxed `y` . With that preparation complete, the operation can be performed: `u.&.>/\.` applies `u.&.>` (in words: unbox each operand, apply `u.`, and rebox) starting at the end of the reversed `y` . The first application of `u.&.>` will be between the original first element of `y` and the initial value; the next application will be between the original second element of `y` and the result of the first application; and so on. The results from the applications of `u.&.>` are collected in an array. Finally, the preparation steps are undone, discarding the initial value, reversing the array into original order, and unboxing the result.

The actual implementation of `LoopWithInitial` is a bit more elegant than that schematic representation. Observe that step 7 is the inverse of step 1, step 6 of 2, and step 5 of 3; each pair is an opportunity to use `u&.v`, and the actual verb produced by `LoopWithInitial` is:

```
   knave LoopWithInitial inity
knave&.>/\.&.(,&(<25 10 100 25 100 50 5 10))&.|.&.(<"_1)
```

which performs all the steps of the longer form. We will examine this conjunction in a later chapter.

You may object that since the left operand of `LoopWithInitial` is applied to each item, it still has to be interpreted for each item, so nothing is gained by avoiding the loop. An astute observation, but in the second part of this book we will learn how to write verbs that are not reinterpreted for each item.

Finally, you may observe that the temporary vector, which only needs to be a list, turns into a rank-2 array by the time we have passed it through each item. Doesn't that waste a lot of space? Yes, it does, and if your problem is big enough that the space matters, you may have a valid application for a loop. An alternative would be to make the temporary vector a public variable accessed by `knave` in the same way that temporary variables would be used in C.

---

26. Loopless Code VII—Sequential Machines

J provides a primitive to handle one class of programs, ill-suited for parallel processing, that can be described systematically: sequential machines. Dyad `;:` takes a sequential-machine description in `x` and a stream of input in `y`, and produces the result called for the the machine description.

A brief overview is as follows. The ***output array*** is initialized to empty. An initial ***row number*** (also called a state number) is chosen. Each item of input is converted into a ***column number***. The column numbers are processed, one per ***iteration***. In an iteration, the next column number is supplied to the ***row/action table***: the row number and column number specify an item of the table, which is a 2-element vector giving the new row number and an ***action code***. The action is performed and the row number is updated, completing the iteration. Execution continues until a 'quit' action is encountered or all the column numbers have been supplied. At that point the output array, which contains the accumulated results of the actions performed, becomes the result of the verb.

To illustrate the use of dyad `;:` we will build a machine to recognize hex constants in character input. A hex constant will be '`0x`' followed by any positive number of hexadecimal digits; we will extract the digits and discard the '`0x`'.

The `x` argument to dyad `;:` is a boxed list `f;s;m;ijr` . `m` and `ijr` may be omitted.

`m` controls the conversion of the items of `y` to column numbers. In the general case, `m` is a list of boxes; then the column number produced for an item of `y` is the index of the first box of `m` whose contents have an item equal to the item of `y` (if there is no such box, a column number of `#m` is used). If `y` is a string, `m` may be a numeric list containing one column number for each character code and the column numbers are `(a. i. y) { m` . If `y` is numeric, `m` may be empty or omitted, and `y` specifies the column numbers directly.

In our example problem, we see that the input characters are in 4 classes: the character `0`; the character `x`, the hexadecimal characters `0123456789abcdefABCDEF`, and all other characters. We will assign these column numbers 3, 2, 1, and 0 respectively. The conversion control `m` can be generated by the statements

```
   m =. a. e. '0x123456789abcdefABCDEF'
   m =. m + a. e. '0x'
   m =. m + a. e. '0'
```

and can be verified by

```
   (a. i. '0x2aq') { m
3 2 1 1 0
```

`ijr` gives the initial values of 3 variables used by the sequential machine: the ***input pointer*** *i*, the ***word pointer*** *j*, and the row number *r*. The input pointer is the index in **`y`** of the next item to be processed, incremented by 1 at the end of each iteration. The word pointer is the index in **`y`** of the first item in the ***current word***; when the action code calls for producing output, the output will start with item *j*. When *j* is **`_1`**, there is no current word. The row number, as noted above, is used to index the row/action table. If **`ijr`** is omitted, it defaults to **`0 _1 0`**, which means start processing the first item of **`y`**, with no current word, and starting in row number 0. This default is acceptable for our example problem.

`s` gives the row/action table. This table has as many rows as needed to encode all the states of the sequential machine, and as many columns as there are possible columns numbers of mapped input. Each 1-cell of **`s`** is a 2-element list. The first element will become the row number for the next iteration. The second is the action code, indicating the action to be performed. The action code is a number from 0 to 6, with meanings as follows:

Action code	Addition to output array	Change to *j* (after any addition to the output)
0	none	none
1	none	*j* =. *i*
2	add single word	*j* =. *i*
3	add single word	*j* =. _1
4	add multiple words	*j* =. *i*
5	add multiple words	*j* =. _1
6	stop—no further iterations are performed	

Executing an ‘add’ action when *j* is **`_1`** produces a domain error.

The action code indicates when a value is appended to the output array. The value that is appended depends on the **`f`** parameter (which came from the **`x`** argument to dyad **`;:`**)

and the values of the iteration variables. The values appended for different values of `f` are (*r*=row number, *c*=column number, *j*=word pointer, *i*=input pointer):

`f`	Value appended	Description
0	the items of `y` between *j* and *i*-1, boxed	Boxed word of `y`
1	the items of `y` between *j* and *i*-1	Unboxed word of `y`
2	*j* , *i*-*j*	Index and length of word
3	*c* + *r* * number of columns in `s`	Coded row and column
4	*j* , (*i*-*j*) , *c* + *r* * number of columns in `s`	Index and length of word, and coded row and column
5	*i* , *j*, *r* , *c* , (<*r*,*c*)`{s`	Input pointer, word pointer, row, column, new row, action

The indicated data is appended to the output array whenever an ‘add single word’ action is executed. The ‘add multiple words’ action also adds the data to the output except that a sequence of consecutive ‘add multiple words’ actions executed from the same row causes only a single item to be added to the output array. The ‘add multiple words’ actions executed after the first modify the item that was added by the first, appending the new word within the previously appended item.

After the last iteration, if no ‘stop’ action has been encountered and *j* is not `_1`, one final ‘emit multiple words’ action is performed.

We can build the state table `s` for our sample problem now. There will be 4 rows. First, we wait for `0`; then we expect `x`; then we expect a hexadecimal digit, signaling start-of-word if we see it; then we wait for a non-hexadecimal-digit, and output the word when we get one. The state table will look like this:

Row number	Description	Column 0 other	Column 1 hexdigit	Column 2 `x`	Column 3 `0`
0	Waiting for `0`	`0 0`	`0 0`	`0 0`	`1 0`
1	Expecting `x`	`0 0`	`0 0`	`2 0`	`0 0`
2	Expecting first digit	`0 0`	`3 1`	`0 0`	`3 1`

3	Waiting for nondigit	0 3	3 0	0 3	3 0

This state table is generated by:

```
   s =. 1 4 2 $ 0 0 0 0 0 0 1 0
   s =. s , 4 2 $ 0 0 0 0 2 0 0 0
   s =. s , 4 2 $ 0 0 3 1 0 0 3 1
   s =. s , 4 2 $ 0 3 3 0 0 3 3 0
```

and we use it with

```
   (0;s;m) ;: 'qqq0x30x30x40x0xxxx'
+--+--+
|30|40|
+--+--+
   (0;s;m) ;: 'qqq0x30x30x40x0x34a'
+--+--+---+
|30|40|34a|
+--+--+---+
```

<<   >>   Contents   Help

27. Modifying an array: `m}`

Modification of a portion of an array, performed in C by the simple expression **x [m] = y**, fits uneasily into J. To begin with, 3 pieces of information are needed: the array `x`, the index `m`, and the replacement data `y` . Since J verbs take only 2 operands, that means the primitive to modify the array will be an adverb so that its left operand can hold one of the 3; the verb derived from the adverb and its operand will perform the modification.

Moreover, the adverb cannot be a modifier of the array's name as **[m]** is a modifier of `x` in **x[m] = y**. In J, the array `x` may not have a name. While in C every array is declared and named, in J we summon up anonymous arrays in the blink of an `i.`, use them, and let them disappear. We expect modification of a subarray to be like other primitives in this regard, which means that it must **not** be a form of assignment to a name using a copula (`=.` or `=:`), but instead a (derived) verb operating on an array to produce a result.

What will the result of the assignment be? This is not much of an issue in C. There the result of the assignment is **y**, but it is seldom used: in C, after we have employed **x[m] = *expression*;** we are usually content to put down the pencil, satisfied that we have described a statement's worth of computation. In J we routinely use weapons of larger bore: after we have modified the array we expect to be able to sort it, or add it to another array, or the like. It is clear that the result of the modification of the array must be the entire modified array.

This reasoning justifies J's design. `}` is an adverb. Modification of an array is performed by dyad `m}` which produces a derived verb of infinite rank. `x m} y` creates a copy of `y` and installs the atoms of `x` into the positions selected by `m` . `m` specifies portions of `y` in the same way as the left operand of `m{y` . Even though dyad `{` has left rank `0`, `m` may have any shape: the atoms of `m` are processed to accumulate a list of selected portions of `y` . `x` must either have the same shape as the selected portion(s) `m{y` or have the same shape as some cell of `m{y` in which case `x` is replicated to come up to the shape of `m{y` . Examples are a recap of the forms of the left operand of dyad `{`, with a couple of twists:

```
   0 (<1 1)} 4 4 $ 5
5 5 5 5
5 0 5 5
5 5 5 5
5 5 5 5
```

The simplest example. We select an atom and modify it.

```
   0 1 2 3 (0)} 4 4 $ 5
0 1 2 3
5 5 5 5
5 5 5 5
5 5 5 5
```

Here we are modifying a selected row.

```
   0 (0)} 4 4 $ 5
0 0 0 0
5 5 5 5
5 5 5 5
5 5 5 5
```

`x` is shorter than `m{y` but `x` can be replicated to match the shape of `m{y` .

```
   0 1 (0)} 4 4 $ 5
|length error
|   0 1    (0)}4 4$5
```

Here the shape of `x` is `2` while the shape of `m{y` is `4`, so the operands do not agree.

```
   1 2 (1 1;2 2)} 4 4 $ 5
5 5 5 5
5 1 5 5
5 5 2 5
5 5 5 5
```

Here `m` has 2 atoms, each selecting a single atom of `y` . `x` has two atoms and they are stored into the selected positions.

```
   1 2 (1 1;1 1)} 4 4 $ 5
5 5 5 5
5 2 5 5
5 5 5 5
5 5 5 5
```

The same element is modified twice. As it happens, the modifications are performed in order, and the last one survives in the result. **Do not rely on this behavior!** As a general rule in J, the order in which a parallel operation is performed is undefined and may change from release to release or from machine to machine.

The interpreter may incorrectly modify the first atom in the array if the atoms of **`m`** select portions of **`y`** that do not have the same shape. You should avoid such **`m`** :

```
   0 1 2 3 (2;1 1)} 4 4 $ 5
3 5 5 5
5 0 5 5
0 1 2 3
5 5 5 5
```

Don't even try to figure out what happened; avoid mixed **`m`** .

Monad `I.`—Indexes of the 1s in a Boolean Vector

If you have a Boolean list with 1s representing items to be modified, you will need to create the list of indexes of the 1s (for selection you would just use **`x # y`**, but for modification you must use **`m}`** which needs the indexes rather than the Boolean list).

Monad **`I.`** (rank 1) performs this function. **`I. y`** produces **`y # i. # y`**, for example:

```
   I. 1 0 0 1 0 1 1
0 3 5 6
```

indicating where the 1s are.

You should use **`I.`**, rather than any equivalent phrase, to perform this function, because the interpreter recognizes compounds such as **`I.@:>`** and handles them with special fast code.

Modification In Place

It is fundamental to the design of J that, in general, verbs produce output in a separate memory block from the inputs. Thus, if I execute **`>: y`**, the result will be in a different memory area from **`y`** . This takes more *space* than incrementing **`y`** in place, but not much additional *computation*, since every atom of **`y`** must be visited and incremented either way. Usually the extra memory usage is immaterial to overall performance. Sometimes we regret having to make a new copy for the output, for example when we just remove the end of a long list with **`}: y`** or add a single character to the end of a long string with **`x , ' '`** .

Dyad **`m}`**'s profligacy with memory bandwidth reaches an extreme: even if **`y`** is huge and only one atom is modified, the whole **`y`** must be copied to produce the result. Sometimes this can be a noticeable drag on performance. To help out in those cases, the interpreter recognizes the specific forms

```
name =. x m} name
name =: x m} name
name =. name , x
name =: name , x
```

and executes the modification in place, i. e. without creating a new copy of ***`name`*** . For the modification to be in-place, there must be nothing else in the sentence either before or after the form shown above, and the two appearances of ***`name`*** must be identical. **`x`** and **`m`** may be any expressions that evaluate to nouns.

<<   >>   Contents   Help

28. Control Structures

Your reward for persevering through two dozen chapters on J is to be shown the direct equivalent of **if/then/else**, **for**, and **while**. I have waited until I am sure you realize that you don't need them, and will choose them not because you see no other way to solve a problem, but because you think they are the best way.

`if.`, `while.`, and the rest are classified as ***control words*** in J. Their part of speech is 'punctuation', like a parenthesis or `LF` character. They are allowed only inside definitions, that is to say inside right operands of the `:` conjunction. If you want to use them from the keyboard or in a script, you must define a verb/adverb/ conjunction and then execute it. A control word ends any sentence to its left, as if the control word started with an end-of-line character. We will cover only a few important control sequences here; to find the rest, bring up the J Vocabulary by pressing F1 and click on Controls which is hidden in plain view next to the heading "Vocabulary".

`for.`/`do.`/`end.` and `for_x.`/`do.`/`end.`

The allowed forms are:

`for.` ***`T-block`*** **`do.`** ***`block`*** **`end.`**
`for_`*`x`*`.` ***`T-block`*** **`do.`** ***`block`*** **`end.`**

The ***`T-block`*** is evaluated and its result **`A`** (that is, the result of the last sentence in the ***`T-block`***) is saved. ***`block`*** is executed once for each item (item, not atom) of **`A`** . If you use the handy **`for_`*`x`*`.`** form (where ***`x`*** represents any valid name of your choice), the private variables ***`x`*** and ***`x`*`_index`** are created, and every time ***`block`*** is executed, ***`x`*`_index`** is assigned the index of an item of **`A`** and ***`x`*** is assigned ***`x`*`_index { A`** .

The **`break.`** and **`continue.`** control words do what you would expect.

`while.`/`do.`/`end.` and `whilst.`/`do.`/`end.`

The allowed forms are:

`while.` ***`T-block`*** **`do.`** ***`block`*** **`end.`**

whilst. ***T-block*** **do.** ***block*** **end.**

while. corresponds to **while** and **whilst.** corresponds to **do while**. The **'st'** in **whilst.** stands for 'skip test' (the first time through), so you get one free pass through the loop, just as with **do while**.

The **break.** and **continue.** control words do what you would expect.

if./do./else./end., if./do./elseif./do./end.

The allowed forms (with optional components underlined) are:

if. ***T-block*** **do.** ***block*** **else.** ***block*** **end.**

if. ***T-block*** **do.** ***block*** **elseif.** ***T-block*** **do.** ***block…*** **end.**

The flow of control is as you would expect. ***T-block***s and ***block***s are both sequences of zero or more J sentences. The result of the last sentence in a ***T-block*** provides the result of the ***T-block*** . The result of a ***T-block*** tests true if its first atom is nonzero, or if it is empty, or if the ***T-block*** was empty. The flow of control is as you would expect based on the tests. The sequence **elseif.** ***T-block*** **do.** ***block*** may be repeated as often as desired, but you will be surprised to learn that once you code an **elseif.** you are not allowed to use **else.** in the same control structure: use **elseif. 1** instead.

My antipathy for **for.** and **while.** has scarcely been concealed, but I harbor no ill will toward **if.** . As long as you don't apply it in a loop, **if.** makes the structure of code obvious and you may use it without remorse. Examples are legion; the form

```
if. # name =. sentence to create an array
  code to process name, which is now known to have items
end.
```

is the most common in my code, used to guarantee that a block of code is executed only when a certain noun has a nonzero number of items.

When we first learned about verb definitions we said that the result of a verb was the result of the last sentence executed. Now we must emend that statement: sentences in ***T-block***s do not affect the result of the verb.

try./catch./end. and catcht./throw.

The simplest form (see the Dictionary for others) is:
try. ***block1*** **catch.** ***block2*** **end.**

try./**catch.**/**end.** is the control-structure equivalent of **u ::v** . ***block2*** is executed only if there was an error during the execution of ***block1*** .

If you want to signal an error, execute the foreign **13!:8 y** where **y** is the error number you want to signal. You can use this in a **try.** block to transfer execution to the corresponding **catch.** block.

throw., when executed in a **try.** block (or in a function executed by that **try.** block), returns control to the **catcht.** block of the **try.** . Just as an error in execution causes the **catch.** block to be executed, a **throw.** causes the **catcht.** to be executed.

select./case./fcase./end.

The form is:
select. ***T-block0*** **case.** ***T-block1*** **do.** ***block1…*** **end.**

T-block0 is evaluated; then the ***T-block***s of the **case.** control words are evaluated sequentially until one is found that matches the result of ***T-block0***; the following ***block*** is then executed, after which control passes to the sentence following the **end.** .

fcase. is like **case.** except that after the ***block*** of an **fcase.** is executed, control passes to the next ***block*** rather than to the sentence following the **end.** .

A ***T-blockn*** matches the result of ***T-block0*** if the result of ***T-block0*** is an element of the result of the ***T-blockn***. So, a ***T-blockn*** could be **2;3;5** and any of those three values would match it. Before this check is made, each side of the comparison is boxed if it is not boxed already. An empty ***T-blockn*** matches anything.

return.

return. ends execution of the definition that is running. The result of the last sentence not in a ***T-block*** is the result. Example:

if. ***T-block*** **do.** ***return-value*** **return. end.**

assert.

`assert.` ***`sentence`***

`assert.` fails with an **`assertion failure`** error if the result of executing the sentence contains any atoms that are not equal to 1. Note that there is no **`end.`** corresponding to the **`assert.`**, so there may be only a single sentence rather than a ***`T-block`***.

<<   >>   Contents   Help

29. Modular Code

Separating your code into independent modules boils down to segmenting the space of variable names into subspaces that overlap to the extent you want. We will discuss how J handles namespaces, and then see how this corresponds to the **classes** and **objects** provided by C++.

Locales And Locatives

Every public named entity in J is a member of a single ***locale***. Each locale has a ***name*** which is a list of characters. A ***locative*** is a name containing a ***simple name*** (the only kind of name we have encountered so far) and an ***explicit locale***, in one of the two forms ***`simplename_localename_`*** and ***`simplename__var`*** . In the form ***`simplename_localename_`***, ***`localename`*** is the name of the explicit locale; in the form ***`simplename__var`***, the variable ***`var`*** must be a scalar boxed string whose opened contents provide the name of the explicit locale. Examples:

`abc_z_` is simple name **`abc`** and locale **`z`**
`vv =. <'lname'`
`def__vv` is simple name **`def`** and locale **`lname`**

Note that a simple name may contain an underscore; it may not end with an underscore or contain two underscores in a row.

(Note: J makes a distinction between ***named locales*** whose names are valid J variable names not including an underscore, and ***numbered locales*** whose names are strings representing nonnegative decimal integers with no leading zeroes. The difference between the two is small and we will ignore it).

The ***current locale*** is a value kept by J and used to influence the processing of names. We will learn what causes it to change. The current locale is the name of a locale. When J starts, the current locale is set to **`'base'`** .

Assignment

An assignment is ***private*** if it is of the form **`simplename=.value`** and is executed while an explicit definition is running (an explicit definition is the result of the **`:`** conjunction, for example a verb defined by **`3 : 'text'`** or a modifier defined by **`adverb define`**). An entity assigned by private assignment is not part of any locale and is accessible only by sentences executed by the explicit definition that was running when the entity was assigned. The idea is this: there is a pushdown stack of namespaces in which private entities are assigned. When an explicit definition `E` is executed, it starts with a new empty namespace that will be destroyed when `E` finishes. Any private assignments made while `E` is running are made in this private namespace. Any private variables referred to by `E` or by tacitly-defined entities invoked by `E` are taken from this private namespace. If `E` invokes another explicit definition `F`, `F` starts off with its own private namespace and has no access to elements of `E`'s private namespace. When `F` finishes, returning control to `E`, `F`'s private namespace is destroyed. `E` is said to be ***suspended*** while `F` is running.

Assignments that are not private (because they assign to a locative, use `=:`, or are executed when no explicit definition is running) are ***public***. Assignment to a locative creates an entity having the simple name in the locative, residing in the explicit locale given by the locative. A public assignment to a simple name creates the named entity in the current locale. Entities in a locale are not affected by completion of an explicit definition; they have a life of their own and can be referred to by any verb that knows how to reach the locale they are in. The following examples illustrate assignments; the interpretation given is correct if the lines are entered from the keyboard:

```
   simp1 =. 5  NB. public (outside of explicit
definition)
   vb1 =: verb define  NB. public
isimp =. simp1  NB. private, referring to public simp1
simp1 =. 8      NB. private (=. inside definition)
loc1_z_ =. 10   NB. public (locative)
simp2 =: 12     NB. public (=:)
isimp,simp1     NB. result
)
   vb1 ''       NB. execute vb1, see result
5 8
```

Note that `simp1` was set to `8` by the explicit definition. Because this was a private assignment, the public value was not changed:

```
   simp1
5
```

The public value is still `5`, as it was before the explicit definition was executed.

```
   loc1_z_
10
```

```
   simp2
12
```

The other public assignments leave their results in the locale they were assigned to.

```
   isimp
|value error: isimp
```

The entities assigned by private assignment were destroyed when **`vb1`** finished.

Note that the **`load`** verb (which runs scripts) is an explicit definition. Any assignment using **`=.`** executed during **`load`** will be lost. Use **`=:`** to define names in scripts.

Name Lookup

Names and locatives used to refer to entities look just like names appearing as targets of assignments, but there is an additional level of complexity for references. Each locale has a ***search path*** (usually called simply the ***path***) which is a list of boxed locale names. The path is set and queried by the foreign **`18!:2`** which goes by the alias **`copath`** . Examples:

```
   ('loc1';'loc2';'z') 18!:2 <'loc3'
```

Sets the path for locale **`'loc3'`** to **`'loc1'`** followed by **`'loc2'`** (and the obligatory **`'z'`**).

```
   copath <'loc3'
+----+----+-+
|loc1|loc2|z|
+----+----+-+
```

Queries the path for **`'loc3'`** .

Every reference to a name implicitly uses a path. A reference to a locative looks for the simple name in the explicit locale; if the name is not found there, the locales in the path of the explicit locale are examined one by one until the simple name is found (if the name is not found in any locale in the path, it is an undefined name). A reference to a simple name is similar, but first the private namespace of the executing explicit definition (if any) is searched, and only if that search fails are locales searched, starting in the current locale and continuing if necessary in the locales in the current locale's path.

Note that only the path of the starting locale (either the current locale or the explicit locale) specifies the search order. No other paths are used.

Examples of references:

```
   ('loc1';'loc2';'z') 18!:2 <'loc3'
   a_loc1_ =: 'a'
   a_loc2_ =: 'b'
   c_loc3_ =: 'c'
   c_loc2_ =: 'd'
   a_loc3_
a
```

The name was not defined in `'loc3'` so the path was used, and the name was found in `'loc1'` .

```
   a_loc2_
b
```

The value in `'loc2'` can be retrieved if we start the search there.

```
   c_loc3_
c
```

If the value is found in the starting locale, no search is performed.

```
   c_loc1_
|value error: c_loc1_
```

We have not defined a path for `'loc1'`, so `'loc2'` is not searched.

Changing The Current Locale

The current locale can be changed in two ways: explicitly by executing the `cocurrent` verb whose purpose is to change the current locale, and implicitly by executing a verb named by a locative.

`cocurrent y` sets the current locale to `y` . Simple as that. `cocurrent` uses the foreign `18!:4` . **Do not use `18!:4` directly!** It is intended to be used under an alias, and it has side effects.

Executing an entity named by a locative (almost always a verb, but it could be a modifier as well) saves the current locale, changes the current locale to the explicit locale of the locative before starting the entity, and resets the current locale to the saved value when the entity finishes. **Note** that **the entity always runs in the explicit locale of the locative**, even if the search for the name found the entity in some other locale in the search path.

Whenever a named entity finishes execution, the locale is restored to its original value, even if the entity changed the current locale.

Here are examples of actions affecting the current locale:

```
   load 'printf'
   18!:5 ''
+----+
|base|
+----+
```

This is how you query the name of the current locale. Next we define two verbs.

```
   v1_z_ =: verb define
'Locale at start of v1 is %j' printf 18!:5 ''
qprintf 'n1 '
v2_result =. v2_loc1_ n1
'Value returned by v2 is %s' printf <v2_result
'Locale in v1 after calling v2 is %j' printf 18!:5 ''
qprintf 'n1 '
)
   cocurrent <'loc2'
   v2 =: verb define
'Locale at start of v2 is %j' printf 18!:5 ''
qprintf 'n1 y. '
cocurrent <'loc2'
qprintf 'n1 '
'Locale at end of v2 is %j' printf 18!:5 ''
n1
)
```

The verb **`v1`** was defined in locale **`'z'`** because it was an assignment to a locative; the verb **`v2`** was defined in locale **`'loc2'`** because it was an assignment to a simple name and the current locale at the time of its assignment was **`'loc2'`** .

```
   cocurrent <'loc3'
   v2 =: [:
```

Now the verb **`v2`** is defined in both locale **`'loc2'`** and **`'loc3'`** . Next we define the noun **`n1`** in each of our locales, so we can see which locale a name was found in:

```
   n1_loc1_ =: 'n1 in loc1'
   n1_loc2_ =: 'n1 in loc2'
   n1_loc3_ =: 'n1 in loc3'
```

Now run the verbs. I will insert interpretation of the execution.

```
   v1 ''
```

J searches for the simple name **`v1`** in the current locale **`'loc3'`**; not finding it there it looks in **`'loc1'`**, **`'loc2'`**, and **`'z'`**, finally finding it in **`'z'`** . J executes the definition of the verb found in `'z'`, but without changing the current locale.

```
Locale at start of v1 is loc3
```

Yes, the current locale is still `'loc3'` …

```
n1=n1 in loc3
```

…and a name lookup uses the current locale as the starting point.

```
Locale at start of v2 is loc1
```

`v1` has executed `v2_loc1_` . J starts searching for the name `v2` in locale `'loc1'` and its path, eventually finding it in `'z'` . `v2` is executed, using the definition found in `'z'`, but with the current locale set to the explicit locale of the locative, namely `'loc1'` . Note that `v2` was also defined in `'loc3'` (as `[:` which would give an error), but `'loc3'` was never searched. The operand of `v2` was `n1`; note that the lookup for `n1` is completely independent of the lookup for `v2`; `n1` is sought and found in `'loc3'` and it is that value that becomes `y.` at the start of execution of `v2` .

```
n1=n1 in loc1 y.=n1 in loc3
```

Simple name lookups start in the current locale `'loc1'` . The private name `y.` has the value it was given on entry.

```
n1=n1 in loc2
```

Here we have switched the current locale to `'loc2'` using `cocurrent`, and the name is found in the new current locale.

```
Locale at end of v2 is loc2
Value returned by v2 is n1 in loc2
```

Here `v2` has finished and control has returned to `v1` . Note that the value returned by `v2` is simply the result of the last sentence executed; it is a noun. Here it is the value of `n1` at the end of `v2`, at which time the current locale was `'loc2'` .

```
Locale in v1 after calling v2 is loc3
```

Note that when `v2` finished the current locale was restored to its value before execution of `v2` . The `cocurrent` in `v2` has no effect once `v2` finishes.

```
n1=n1 in loc3
```

Execution of the verb `v1` is complete. We should be back in locale `'loc3'` from which we executed `v1` :

```
   18!:5 ''
+----+
|loc3|
+----+
```

The Shared Locale `'z'`

It is a J convention, universally adhered to, that every locale's search path must end with the locale **`'z'`** . Any name in the **`'z'`** locale can then be referred to by a simple name from any locale, making names in the **`'z'`** locale truly global names.

Using Locales

You have my sympathy for having to read through that detailed description of name processing; I refuse to apologize, though, because you really can't write programs if you don't know what names mean. But, you wonder, How do I *use* locales?

You won't go far wrong to think of a locale as akin to a **`class`** in C++. When you have a set of functions that go together as a ***module***, define all their verbs and nouns in a single locale. The easiest way to do this is to put a line
`coclass <'`*`localename`*`'`
at the beginning of each source file for the module (**`coclass`** is like **`cocurrent`** but it supports inheritance). Then, every public assignment in the file will automatically be made in the locale ***`localename`*** .

The names defined in the locale are the equivalent of the private portion of class. To provide the public interface to the class, you need to put those names in a place where they can be found by other modules. The traditional way to do this is to define them in the locale **`'z'`** by ending each file with lines like
`epname_z_` **`=:`** ***`epname_localename_`***
Here ***`epname`*** is the name of a verb, and ***`localename`*** is the name of the module's locale. Take a minute to see what happens when some other locale invokes ***`epname`*** . The name search for ***`epname`*** will end in the locale **`'z'`** where this definition is found. Execution of this definition immediately results in execution of ***`epname_localename_`*** which switches the current locale to ***`localename`*** and runs the definition of ***`epname`*** found there. The benefit is that the calling module doesn't need to know the locale that ***`epname`*** is going to be executed in.

If you tire of writing out the public definitions one by one, I have included in **`jforc.ijs`** a verb to do it for a list of entry points using a sentence like
`PublishEntryPoints 'public1 public2 public3'`

If you want to create multiple copies of objects derived from a class, you should consult the Lab on Object Oriented Programming. There you will learn about numbered locales and how to create and destroy objects. We will not discuss these topics here.

By following the guidelines given above you will be able to emulate the class facilities of C++. Because J is interpreted, you can do much more if you want: you can change search paths dynamically; you can use locales and paths to create a high-performance network database; you can pass locales as data and use them to direct processing; you can peek at a module's private data. You can even modify a module's private data from outside the module, but if you are struck by lightning after doing so the coroner will find it was suicide.

Using locale-names as data allows for dynamic separation of namespaces. For example, the processing of forms in J requires definition of verbs for a great many events. You may let these verbs all share the same locale; but if you want to segregate them, the Window Driver will remember what locale was running when each form was displayed, and direct events for a form to the locale handling the form.

---

30. Writing Your Own Modifiers

If you find that you are coding recurring patterns of operations, you can write a modifier that represents the pattern. You will find that reading your code is easier when the patterns are exhibited with names of your choosing.

You write a modifier like you write a verb, using `conjunction define` or `adverb define`, or `2 :n` or `1 :n` for one-liners. When you assign the modifier to a name, that name becomes a conjunction or adverb, and it will be invoked as `u` *`name`* `v y` (monad) or `x u` *`name`* `v y` (dyad) if it is a conjunction, or `u` *`name`* `y` (monad)or `x u` *`name`* `y` (dyad) if it is an adverb.

When a modifier is invoked, the lines of the modifier are executed one by one, just as when a verb is invoked, and the result of the last sentence executed becomes the result of the modifier. The `u` (and `v`, for conjunctions) operand(s) of the modifier are assigned to the local names `u.` (and `v.`) when the modifier starts execution (in addition, if `u` is a noun, it is assigned to the local name `m.` and if `v` is a noun it is assigned to `n.`)

User-written modifiers are of two types: those that refer to the variables `x.` and `y.`, and those that do not.

Modifiers That Do Not Refer To `x.` Or `y.`

If the modifier does not refer to `x.` or `y.`, its text is interpreted when its operands (`u` and, for conjunctions, `v`) are supplied, and its result is an entity which may be any of the four principal parts of speech. The result replaces the modifier and its operands in the sentence, and execution of the sentence continues.

Usually you will want to create a verb, but nothing keeps you from writing a conjunction whose result is another conjunction. Here we will confine ourselves to verb results.

If a modifier does not refer to `x.` or `y.`, it can be invoked without any `x.` or `y.`; only the `u.` (and `v.`, for conjunctions) are used. The text of the modifier is executed and the resulting verb replaces the modifier and its operands in the execution of the sentence.

Let's write some of the utility modifiers referred to in earlier chapters. **`Ifany`** was an adverb that executed **`u`** if **`y`** had a nonzero number of items:

```
   9!:3 (5)  NB. Do this once to select simplified
display
   Ifany =: 1 : 'u. ^: (*@#@])'
   < Ifany
<^:(*@#@])
```

`Ifany` does not need to look at **`y.`**; it creates a verb that executes **`u`** only if **`y`** has items. Here we have executed the adverb **`Ifany`** with the left operand **`<`**, and the result is a verb—the compound verb **`<^:(*@#@])`** . We can execute that verb on a noun operand:

```
   < Ifany 1 2 3
+-----+
|1 2 3|
+-----+
```

Remember that **`Ifany`** is an adverb, so it has precedence and the line is executed as if **`(< Ifany) 1 2 3`** . The verb **`(< Ifany)`**, which has the value **`<^:(*@#@])`**, is applied to **`1 2 3`** and produces the boxed result.

```
   < Ifany ''

```

An empty **`y`** is left unboxed.

`u Butifnull n` was a conjunction that applied **`u`** if **`y`** had items, otherwise it produced a result of **`n`** . It could be written:

```
   Butifnull =: 2 : 'n."_ ` u. @. (*@:#@:])'
```

Again **`(*@:#@:])`** will check whether **`y`** has items, and this time the result will be used to select the appropriate verb to execute.

```
   < Butifnull 5
5"_`<@.(*@:#@:])
```

When **`Butifnull`** is executed with operands, it produces a verb.

```
   < Butifnull 5  'abc'
+---+
|abc|
+---+
   < Butifnull 5  ''
5
```

The verb it produces can be applied to its own noun operands.

Example: Creating an Operating-System-Dependent Verb

The great thing about modifiers that do not refer to `x.` or `y.` is that they are fully interpreted before the `x` and `y` operands are supplied, so there is no interpretive overhead during the processing of the data. Here is a more complex example taken from the J system. The goal is to define a verb `playsound` that can be used to play a .wav file under Windows:

```
NB. y. is the file data to be played
playsound =: '' adverb define
select. 9!:12 NIL
case. 2 do.
  'winmm.dll sndplaysound i *c i' & (15!:0) @ (;&1)
case. 6 do.
NB. 2=nodefault + 4=memory  +  16b20000 = file
  'winmm.dll PlaySound i *c i i' & (15!:0) @ (;&(0;4))
end.
)
```

To begin with, let's make sense of this odd sequence `'' adverb define` . The `adverb define` defines an adverb, but what's the `''`? Simple—it's the left argument to the adverb that was defined: the adverb is executed with `u.` set to `''` . The result of that *execution* of the adverb is what gets assigned to `playsound` .

So, what happens when the adverb is executed? The adverb calls the foreign `9!:12` to see what operating system is running, and executes a selected line that contains the definition of a compound verb. Since that line is the last one executed, it becomes the result of the adverb; so the result of the adverb is the selected verb, and that is what is assigned to `playsound` . On my system, this leaves `playsound` defined as a single compound verb:

```
   playsound
'winmm.dll PlaySound i *c i i'&(15!:0)@(;&(0;4))
```

Lovely! No check for operating system needs to be made when I invoke `playsound`; the check was made when `playsound` was defined.

Example: The `LoopWithInitial` Conjunction

The conjunction `LoopWithInitial` that we learned about earlier can be written as

```
   LoopWithInitial =: 2 : 'u.&.>/\.&.(,&(<v.))&.|.&.(<"_1)'
```

It's just one application of `&.` after another. We can use it to illustrate a subtlety about modifiers that you should be aware of. Consider an invocation of `LoopWithInitial` :

```
   vb =. +
   init =. 4 5
   vb LoopWithInitial init
vb&.>/\.&.(,&(<4 5))&.|.&.(<"_1)
```

The verb that is produced seems in order, but notice one point: the verb contains the value of `init`, but the name of `vb` . This is a rule: **the name of a verb argument is passed into a modifier, but the value of a noun argument is passed**. Note that if these lines appear inside a verb, the verb `vb`, which is assigned by private assignment, is not defined inside `LoopWithInitial`, because `LoopWithInitial` is running in a different explicit definition from the one in which `vb` was assigned. As we see above, `LoopWithInitial` can pass `vb` into other modifiers, but if `LoopWithInitial` tried to execute `vb` it would fail.

Before we move on I want to point out one tiny example of the beauty of J. For `&.(,&(<4 5))` to work, there must be some obverse of `,&(<4 5)` that undoes its effect. What would that be? We can see what the interpreter uses:

```
   ,&(<4 5) b. _1
}: :.(,&(<4 5))
```

It undoes the addition of a trailing item with `}:` which discards the last item. Yes, that makes sense (the obverse has its own obverse which is the original verb).

Example: A Conjunction that Analyzes `u` and `v`

The conjunction `u&.v` expresses with great clarity the sequence of applying a transformation `v`, then applying the operation `u`, then inverting the transformation `v` . The dyad `x u&.v y` applies the same transformation to both `x` and `y`, but in many cases the transformation is meaningful only on one operand, and what we would like is a conjunction `Undery` such that `x u Undery v y` produces `v^:_1 x u v y` . For example, to encipher the characters of `y` by replacing each one by the letter `x` positions earlier, we would use

```
   5 1 3 2 -~ Undery ('abcdefghijkl'&i."0) 'hijk'
```

to perform the function

```
   t =. 'abcdefghijkl'&i."0 'hijk'
   t =. 5 1 3 2 -~ t
   t { 'abcdefghijkl'
chgi
```

With that `x` stuck in the middle of the desired result `v^:_1 x u v y` it appears that we will have to refer to `x.` in our conjunction, but actually we can use an advanced feature of J to make the `x.` disappear. The sequence `(u v)` produces a verb that, when executed as the dyad `x (u v) y`, gives the result of `x u v y` (you will learn about this and more if you persevere with the part of the book devoted to tacit programming). So, the verb we are looking for is `v^:_1 @: (u v)` and we can write

```
   Undery =: 2 : 'v.^:_1 @: (u. v.)'
   5 1 3 2 -~ Undery ('abcdefghijkl'&i."0) 'hijk'
chgi
```

Before we pat ourselves on the back for this achievement, we should consider whether the verb produced by `Undery` has the proper rank. We see that it does not: `Undery` applies `v` to the entire `y`, and `u` to the entire `x` and the result of `u y`, when really we should be performing the operation on *cells* of `x` and `y`, where the cell-size of `x` is given by the left rank of `u` and the cell-size of `y` is given by the right rank of `v` . For example, if we wanted to take the `-x` least-significant bits of `y`, we could use

```
   _3 {."0 1 Undery #: 30
12
```

(remember that monad `#`: converts an integer `y` to its binary representation, producing a Boolean list–we are taking `x` bits of that and then converting back to integer) The binary code for 30 is `11110`, the 3 low-order bits are `110`, and the result is `6`. But when we have list arguments, we get an incorrect result:

```
   _3 _4 _3 {."0 1 Undery #: 32 31 30
0 15 12
```

The result for `30` is wrong: because `#:` was applied to the entire `y`, `110` was extended with framing fills to become `1100`, and the result is `12` instead of the expected `6`. To get the right result we need to apply the verb to cells of the correct size:

```
   _3 _4 _3 ({. Undery #:"0) 32 31 30
0 15 6
```

and naturally we would like to make `Undery` automatically produce a verb with the correct rank.

The way to find the rank of the verb `u` is to execute `u b. 0` . In our conjunction `u.` and `v.` are verbs, and we can use their ranks to produce an `Undery` that gives the correct rank:

```
   Undery =: 2 :'(v.^:_1)@:(u. v.)"((1{u.b.0),2{v.b.0)'
```

We have selected the left rank of `u` and the right rank of `v`, and put them as the ranks of the verb produced by `Undery` . This produces the desired result:

```
   _3 _4 _3 {."0 1 Undery #: 32 31 30
0 15 6
```

and we can see the verb produced by `Undery`, with its ranks:

```
   {."0 1 Undery #:
#:^:_1@:({."0 1 #:)"0 0
```

This version of `Undery` produces correct results, but we should add one small improvement: the inverse of monad `#:` should be monad `#.` rather than monad `#:^:_1`, because the two forms are different. One difference is obvious: the rank of `#:^:_1` is infinite, while the rank of `#.` is 1; but that is immaterial in `Undery` . The other difference is subtle but it could be significant: the two forms may have different performance in compounds. The interpreter recognizes certain compounds for special handling; the list grows from release to release, but it's a pretty safe bet that `#.` will be selected for special treatment before `#:^:_1` (and `<` before `>^:_1`, and so on). So, we would like to replace the `v.^:_1` with the actual inverse of `v` . We can get the inverse of `v` by looking at `v b. _1` which produces a character-string representation of the inverse of `v` . We can then convert this string to a verb by making it the result of an adverb (we can't make it the result of a verb, because the result of a verb must be a noun). So, we are led to

```
  Undery=:2 :'(a: 1 :(v.b._1))@:(u.v.)"((1{u.b.0),2{v.b.0)'
```

where we defined the adverb `1 :(v.b._1)` and then immediately executed it with an ignored left operand `a:` to create the desired verb form. Now we have

```
   {."0 1 Undery #:
#.@:({."0 1 #:)"0 0
```

which we can be content with.

An Exception: Modifiers that Do Not Refer to `u.` or `v.`

In very early versions of J, modifiers could not refer to their `x.` and `y.` operands.

In those days, a modifier used the names **`x.`** and **`y.`** to mean what we now mean by **`u.`** and **`v.`** . Modern versions of J continue to execute the old-fashioned modifiers correctly by applying the following rule: if a modifier does not contain any reference to **`u.`**, **`v.`**, **`m.`**, or **`n.`**, it is assumed to be an old-style modifier, and references to **`x.`** and **`y.`** are treated as if they were **`u.`** and **`v.`** . You may encounter old code that relies on this rule, but you should not add any new examples of your own.

Modifiers That Refer To `x.` Or `y.`

Most of the modifiers you write will be refer to **`x.`** and **`y.`** . The names **`x.`** and **`y.`** refer to the noun operands that are supplied when the modifier is invoked as **`[x] u`** ***`adverb`*** **`y`** or **`[x] u`** ***`conjunction`*** **`v y`** .

Here is an example, which is invoked as **`[x] u InLocales n y`**, where **`u`** is a verb and **`n`** is a list of locale names; it executes **`u y`** (or **`x u y`** if the invocation is dyadic) in each locale of **`n`** :

```
InLocales =: 2 : 0
l1 =. 18!:5 ''
for_l. n. do.
  cocurrent l
  u. y.
end.
cocurrent l1
''
:
l1 =. 18!:5 ''
for_l. n. do.
  cocurrent l
  x. u. y.
end.
cocurrent l1
''
)
```

This illustrates the important points. The text of the definition is not interpreted until the **`x.`** and **`y.`** are available, in other words until the verb *defined* by **`u InLocales n`** is invoked. Since that invocation may be either monadic or dyadic, two versions of the conjunction are given, one for each valence. The result of the execution must be a noun, because the definition defines a verb and the result of a verb is always a noun.

That last point is important and I want to emphasize it. It is true that `InLocales` is a conjunction, and yet its text defines a verb. How is this possible? Because **`InLocales` is executed as a conjunction at the time it gets its `u` and `n` operands, but its text is not interpreted until the derived verb (which consists of `u`, `n`, and the text of `InLocales`) gets its `x` and `y` operands**. When `InLocales` is supplied with `u` and `n`, it is executed to produce a verb which consists of the text of `InLocales` along with `u` and the value of `n` . This ***derived verb*** is hidden inside the interpreter where it waits to be applied to a `y` (and possibly `x`). When the derived verb is given its operands, it starts interpreting the text of `InLocales` (which was unusable until the time that `y.` and `x.` could be given values) and initializes `u.` and `n.` from the values that were saved when `u InLocales n` was executed. Thus the text of `InLocales` describes a verb operating on `y.` and `x.` .

Your modifiers should refer to `x.` and `y.` only if necessary. `Ifany` from the previous section could have been written

```
   Ifany =: 1 : 'u.^:(*#y.) y.'
```

which would produce exactly the same result as the other definition, but it would usually be slower, because the text could not be interpreted until `x.` and `y.` could be defined. If the conjunction happens to be used in a verb of low rank, the result could be soporific.

Here's a puzzle that may be of interest to those readers whose eventual goal is Full Guru certification. Why did `InLocales` save and restore the current locale? Didn't we say that completion of any named entity restores the original locale?

Let's see what happens when we don't restore, using a simple testcase:

```
   t =: 1 : 0
cocurrent u.
y.
)
   (<'abc') t 0
0
   18!:5 ''
+---+
|abc|
+---+
```

Sure enough, the current locale was changed! But see what happens when we give a name to the verb created by the execution of `t` :

```
   cocurrent <'base'
   tt =: (<'abc') t
   tt 0
0
   18!:5 ''
+----+
|base|
+----+
```

The current locale was restored. What causes the difference?

The answer is that in the sentence `(<'abc') t 0`, the named adverb `t` is executed when it is given its operand `<'abc'` . The result of that execution is the derived verb `(<'abc') t` *which has no name*. When the derived verb is executed with the operand `0`, the text is interpreted, causing a change to the current locale, and when the derived verb finishes, the current locale is not restored because the derived verb is anonymous. If we give that derived verb a name (`tt` here), it restores the current locale on completion.

The observed behavior reinforces the point that the **text** of a modifier that refers to `x.` or `y.` is not interpreted when the modifier is executed; it **is interpreted only when the derived verb is executed**.

<<   >>   Contents   Help

31. Applied Mathematics in J

Complex Numbers

All the mathematical functions can be applied to complex numbers. A complex constant is written with the letter **`j`** separating the real and imaginary parts, e. g. **`0j1`** is the square-root of **`_1`** . Alternatively, a constant can be written in ***polar form*** in which the letters **`ar`** (or **`ad`**) separate the magnitude of the number from the angle in radians (or degrees) between the real axis and a line in the complex plane from the origin to the point representing the number:

```
   1ar1
0.540302j0.841471
   1ad90
0j1
```

A number of verbs are available for operations on components of complex numbers. All have rank 0.

`+ y` conjugate of **`y`**

`+. y` creates a 2-atom list of the real and imaginary components of **`y`**

`*. y` creates a 2-atom list of the length and angle in radians of the polar form of **`y`**

`| y` magnitude of **`y`**

`j. y` **`0j1 * y`**

`r. y` **`^ 0j1 * y`**

`x j. y` **`x + j. y`** (i. e. **`x`** is the real part, **`y`** is the imaginary part)

`x r. y` **`x * r. y`** (i. e. polar form, where **`x`** is the magnitude and **`y`** the angle in radians)

`! y` factorial of **`y`** (more generally, the gamma function Γ(1+**`y`**))

Matrix Operations

J has primitive verbs for operations on matrices, some using the conjunction `.` . `-/ .* y` gives the determinant of `y`; `x +/ .* y` is the matrix product of `x` and `y`; `%. y` is the matrix inverse of `y` (provided `y` is nonsingular); `x %. y` is the projection of `x` onto `y` .

In `x +/ .* y`, a rank-1 `x` is treated as a matrix with one row, and a rank-1 `y` is treated as a matrix with one column; but the result rank, which is 2 when rank-2 matrices are multiplied, is 1 when one operand has rank 1 and 0 when both do.

`x %. y` is a rough-and-ready way to get a least-squares approximation of `x` as a linear combination of the columns of `y` . If your `y` is singular or close to it, avoid `%.` and use methods based on singular value decomposition.

Polynomials: `p.`

J supports 3 different ways of specifying a polynomial:

1. as a list of coefficients of increasing powers, starting with power 0 (i. e. a constant term), where each coefficient multiplied by the corresponding power of the variable produces a term; the polynomial is the sum of the terms. This form is an unboxed numeric list.
2. as a multiplier `m` and a list of roots `r`, representing the polynomial $m(x-r_0)(x-r_1)\ldots(x-r_n)$. This form is a 2-item list of boxes `m;r` (if `m` is 1 it may be omitted, leaving just `<r`).
3. (for multinomials) a multinomial of *n* variables is represented as a list of terms, each term consisting of a coefficient and a list of *n* exponents (one exponent per variable). The multinomial is the sum of the terms. The form is a boxed rank-2 array in which each item is a coefficient followed by the list of *n* exponents. This form is distinguished from the multiplier-root form by the rank of the boxed array. (It is possible to have multiple multinomials that share a common exponent array, by having more than one coefficient preceding each list of exponents, but we will not pursue that here)

For example, three ways of expressing the polynomial $3x^3$-12x (which can be written 3x(x+2)(x-2))are `0 _12 0 3`, `3;_2 0 2`, and `<2 2$_12 1 3 3` .

`p. y` has rank 1 and converts between coefficient and multiplier-root form of the polynomial `y` . Note that converting from coefficient form to multiplier-root form solves for the roots of the polynomial.

```
   p. 0 _12 0 3
+-+------+
|3|2 _2 0|
+-+------+
   p. 3;2 0 _2
0 _12 0 3
```

If the multinomial form has only one variable (i. e. each item has length 2), monad **`p.`** will convert it to coefficient form:

```
   p. <2 2$_12 1 3 3
0 _12 0 3
```

Dyad **`p.`** has rank **`1 0`** and is used to evaluate a polynomial in any of these forms. If **`x`** is a polynomial in coefficient or multiplier-root form, **`x p. y`** evaluates it with **`y`** giving the value of the variable:

```
   0 _12 0 3 p. 1
_9
   (3;_2 0 2) p. _2 _1 0 1 2
0 9 0 _9 0
```

(the second evaluation applied the polynomial to 5 different values of the variable).

If **`x`** is a multinomial, **`x p. <y`** evaluates it with the list **`y`** giving the values of the variables (**`y`** must be boxed because the right rank of dyad **`p.`** is 0 and in this case there is more than one variable). So to evaluate the binomial $x^3+3x^2y+3xy^2+y^3$ with $x=2$ and $y=3$ we have

```
   (<4 3$1 3 0  3 2 1  3 1 2  1 0 3) p. <2 3
125
```

as expected.

Calculus: `d.`, `D.`, `D:`, and `p..`

J provides support for differential and integral calculus. **`u d. n`** produces the verb that gives the **`n`**th derivative of **`u`** :

```
   *: d. 1
+:
```

`*: y` is **`y`** squared; the derivative is **`+: y`** which is **`y`** doubled.

```
   ^&3 d. 1
3&*@(^&2)
```

`^&3 y` is `y` cubed; the derivative is `3 * y ^ 2` .

```
   ^&3 d. 2
3"0 * +:
```

Second derivative is `3 * 2 * y` .

```
   *: d. _1
0 0 0 0.33333&p.
```

The _1st derivative is the indefinite integral, which is `(y ^ 3) % 3` . The form the interpreter uses is the polynomial form.

```
   f =. *:
   f d. _1 (6)
72
   g =. *:@+:
   g d. _1 (6)
288
```

`u d. n` produces ordinary derivatives, and evaluates its `u` with rank 0. For partial derivatives, use `u D. n` where `u` has rank greater than 0. Each cell of `y` produces an array of results, one result for each atom of the cell, giving the partial derivative with respect to that atom. For example, the length of a vector is given by

```
   veclength =: +/&.:*:"1
```

which squares the atoms, adds them, and takes the square root:

```
   veclength 3 4 5
7.07107
```

The derivative of the vector length with respect to the individual components is given by:

```
   veclength D. 1 (3 4 5)
0.424264 0.565685 0.707107
```

The result of `u` need not be a scalar. Here we define the cross product:

```
   xp =: dyad : '((1|.x.)*(_1|.y.)) - ((_1|.x.)*(1|.y.))'"1
   0 0 2&xp D. 1 (4 2 0)
 0 2 0
_2 0 0
 0 0 0
```

Each row is the vector-valued partial derivative of the cross product (`0 0 2 xp 4 2 0`) with respect to one component of `y` .

The interpreter will bend every effort to find a derivative for your function, but it may fail, or you may not like the derivative it chooses. `m D. n`, when `m` is a gerund ``u`v``, produces a new verb which executes like `u` but whose n^th^ derivative is `v` :

```
   a =. *:`] D. 1
```

Here `a` is defined to be `*:` except that its derivative is `]` .

```
   a
*:`]D.1
   a D. 1 (5)
5
```

Sure enough, the derivative is `]` .

If you don't want to express the derivative, you can have the interpreter approximate it for you. `x u D: n y` approximates the derivative at `y` by evaluating `u` at `y` and `y+x` . Both the left and right ranks of `u D: n` are the monadic rank of `u`, so you can specify different step-sizes for different atoms of `y` (if `x` is a scalar, it is used for the step-size at all atoms of `y`).

You can do calculus on polynomials by manipulating the polynomial forms without having to create the verbs that operate on those forms. `p.. y` (rank 1) takes polynomial `y` in either coefficient or multiplier-root form and produces the coefficient form of the derivative of `y` . `x p.. y` (rank `0 1`) produces the coefficient form of the integral of `y` with constant term `x` .

Taylor Series: `t.`, `t:`, and `T.`

With derivatives available, Taylor series are a natural next step. `u t. y` is the `y`th Taylor coefficient of `u` expanded about 0, and `x u t. y` evaluates that term at the point `x`, in other words `x u t. y` is `x^y * u t. y` . All ranks of `u t.` are 0.

`u T. n` is a verb which is the n-term Taylor approximation to `u` (expanded about 0).

`u t: y` is `(!y) * u t. y` .

Hypergeometric Function with `H.`

The ***generalized hypergeometric function*** is specified by two lists of numbers, a ***numerator list*** and a ***denominator list***. The generalized hypergeometric function is the sum over all *k* of the (infinite) generalized hypergeometric series, which is a power series in which the coefficient of the y^k term is the product of the rising factorials of length k of the numerator items divided by a similar product for the denominator items, and then divided by `!k` .

The conjunction `H.` is used in the form `m H. n` where `m` is the numerator list and `n` is the denominator list. The resulting verb `m H. n` has rank 0. The monad `m H. n y` takes the limit of the sum of the generalized hypergeometric series; the dyad `x m H. n y` takes the sum of the first `x` terms. Formally, the generalized hypergeometric function is

$$\sum_{k=0}^{\infty} \frac{(m_0)_k (m_1)_k \ldots (m_{<:\#m})_k}{(n_0)_k (n_1)_k \ldots (n_{<:\#n})_k} \frac{y^k}{k!}$$ where $(a)_k = a(a+1)\ldots(a+k-1)$

If `m` contains 2 items and `n` contains 1 item, `m H. n` defines a ***hypergeometric function***.

Generalized hypergeometric functions can be used to calculate a great many functions of interest: Legendre polynomials, Laguerre polynomials, Chebyshev polynomials, and Bessel functions of the first kind are all special cases of hypergeometric functions. Ewart Shaw, in http://www.ewartshaw.co.uk/data/jhyper.doc, gives a number of examples of uses of `H.` . For example, the error function and the cumulative distribution function are given by

```
   erf =: 3 : '(((2p_0.5)*y.) % (^*:y.)) * 1 H. 1.5 *: y.'
   n01cdf =: 3 : '-: >: erf y. % %:2'   NB. CDF of N (0,1)
```

where I have rewritten Shaw's formulas to use elementary J. `2p_0.5` is $2/sqrt(\pi)$.

Sparse Arrays: Monad and Dyad `$.`

`$. y` converts the array `y` into a sparse-matrix representation which can save a lot of space and time if most of the atoms of `y` have the same value. `$.^:_1 y` converts sparse `y` back to normal (***dense***) form. A large but incomplete subset of operations is supported on sparse arrays; look at the description of `$.` if you think you'd like to use them.

Random Numbers: `?`

Monad `?` has rank 0. If `y` is 0, `? y` is a random floating-point number uniformly distributed in the interval 0 <= `? y` < 1. If `y` is positive, `? y` is a random element of `i. y` . An example use is

```
   ? 3 3 $ 1000
755 458 532
218  47 678
679 934 383
```

Dyad **?** has rank 0. **x ? y** is a list of **x** items selected without repetition from **i. y**, as if the list **i. y** were shuffled and the first **x** elements were taken:

```
   5 ? 52
24 8 48 46 22
```

The **?** verbs use Knuth's GB_FLIP generator.

Plot

J includes a great package for making 2-D and 3-D plots. Check out the Lab named Plot Package to see how to use it. For a quick preview, enter

```
load 'plot numeric trig'
'surface' plot sin */~ steps 0 3 30
```

to see how easy it is to get a plot of *sin*(*x***y*).

Computational Addons

The web site at www.jsoftware.com has several ***addons*** that you can download. These are executable libraries, along with J scripts to call functions in them, that offer efficient implementations of often-used functions. Two of interest in applied mathematics are the LAPACK addon and the FFT addon. If you want a fast implementation of the singular value decomposition referred to earlier, install the LAPACK addon; then you can use

```
   require 'addons\lapack\lapack'
   require 'addons\lapack\dgesvd'
   dgesvd_jlapack_ yourmatrix
```

which will quickly return the desired singular values and singular vectors.

Useful Scripts Supplied With J

The directory ***yourJdirectory*/system/packages** contains a number of subdirectories full of useful scripts. The **/math** and **/stats** subdirectories have scripts for mathematics and statistics; other subdirectories cover topics such as finance, printing, graphics, and interfacing to Windows.

32. Elementary Mathematics in J

Verbs for Mathematics

All the verbs have rank `0 0` .

- `x *. y` Lowest common multiple of `x` and `y`
- `x +. y` Greatest common divisor of `x` and `y`
- `x ! y` number of ways to choose `x` things from a population of `y` things. More generally, `(!y) % (!x) * (!y-x)`

The verbs dyad `e.` (Member of) and dyad `-.` (set difference) are useful in working with sets.

Extended Integers, Rational Numbers, and `x:`

In J, numbers are not limited to 32-bit integers or 64-bit floating point. ***Extended integer*** and ***rational*** are atomic data types (like numeric, literal, and boxed) that allow representation of numbers with arbitrary accuracy. An extended integer constant is defined by a sequence of digits with the letter `x` appended; a rational constant is two strings of digits (numerator and denominator) separated by the letter `r`; examples are `123x` and `4r5` .

The various representations of numbers in J can be given a priority order:

boolean (low) - integer - extended integer - rational - floating point - complex (high)

When a dyadic arithmetic operation is performed on operands of different priorities, the lower-priority operand is converted to the higher-priority representation. The simplest example arises in a list constant:

```
   1234567890123456789 4r5
1234567890123456789 4r5
```

The integer was made into a rational number so it keeps its precision.

```
   3.0 4r5
3 0.8
```

The floating-point constant forces the rational number to floating point.

```
   2 * 3r4
3r2
```

The operation was performed on rational operands with a rational result.

Results of verbs are given a higher-priority representation if necessary:

```
   %: 4r9
2r3
   %: 5r9
0.745356
```

Explicit conversions between extended/rational and floating-point can be performed by the infinite-rank verb **`x:`** . **`x: y`** converts floating-point **`y`** to rational or integer **`y`** to extended integer. The inverse, **`x:^:_1 y`**, converts in the other direction.

A rational number can be split into numerator and denominator by **`2 x: y`** (rank 0):

```
   2 x: 1r3 5
1 3
5 1
```

Factors and Primes: Monad `p:`, Monad and Dyad `q:`

`p: y` (rank 0) gives the **`y`**th prime (prime number 0 is **`2`**).

`q: y` (rank 0) gives the prime factors of **`y`**

`x q: y` (rank 0) with positive **`x`** is the first **`x`** items (or all items, if **`x`** is _) in the list of exponents in the prime factorization of **`y`** :

```
   _ q: 700
2 0 2 1
   */ (p: 0 1 2 3) ^ 2 0 2 1
700
```

`x q: y` with negative **`x`** returns a 2-row table. The second row is the nonzero items of **`(|x) q: y`** (i. e. the nonzero exponents in the prime factorization); the first row is the corresponding prime numbers:

```
   __ q: 700
2 5 7
2 2 1
```

Permutations: `A.` and `C.`

In the ***direct representation*** of a permutation **`p`** each item **`i{p`** of the permutation vector indicates the item number that moves to position **`i`** when the permutation is applied. Applying the permutation in the direct form is as simple as writing **`p{y`** .

The ***standard cycle representation*** of a permutation gives the permutation as a list of cycles (sets of elements that are replaced by other elements of the set). The standard cycle form is a list of boxes, one for each cycle, with each cycle starting with the largest element and the cycles in ascending order of largest element.

Monad **`C. y`** (rank 1) converts between direct and standard-cycle representations of the permutation **`y`** :

```
   /: 3 1 4 1 5 9
1 3 0 2 4 5
   C. /: 3 1 4 1 5 9
+-------+-+-+
|3 2 0 1|4|5|
+-------+-+-+
   C. C. /: 3 1 4 1 5 9
1 3 0 2 4 5
```

`x C. y` (rank **`1 _`**) permutes the items of **`y`** according to the permutation **`x`** which may be in either standard-cycle or direct form; other nonstandard forms are also supported as described in the Dictionary.

There are **`!n`** possible permutations on n items, so it is possible to give each one a number between **`0`** and **`<:!n`** . Imagine the table of all possible permutations in lexicographic order; the ***anagram index*** of a permutation is its index in that table. **`A. y`** (rank 0) gives the anagram index for the permutation **`y`**, which may be in either direct or standard-cycle form. **`x A. y`** (rank **`0 _`**) permutes the items of **`y`** according to the permutation whose anagram index is **`x`** :

```
   a =. /: 3 1 4 1 5 9
   A. a
168
   a C. 1 2 3 4 5 6
2 4 1 3 5 6
   168 A. 1 2 3 4 5 6
2 4 1 3 5 6
```

The monad **`C.!.2 y`** gives the ***parity*** of **`y`** : **`1`** if an even number of pairwise exchanges are needed to convert **`y`** to the identity permutation **`i.#y`**, **`_1`** if an odd number are needed, **`0`** if **`y`** is not a permutation.

---

33. Odds And Ends

To keep my discussion from wandering too far afield I left out a number of useful features of J. I will discuss some of them briefly here.

Dyad # Revisited

`x # y` does not require that `x` be a Boolean list. The items of `x` actually tell how many copies of the corresponding item of `y` to include in the result:

```
   1 2 0 2 # 5 6 7 8
5 6 6 8 8
```

Boolean `x`, used for simple selection, is a special case. If an item of `x` is complex, the imaginary part tells how many cells of fill to insert after making the copies of the item of `y` . The fill atom is the usual `0`, `' '`, or `a:` depending on the type of `y`, but the fit conjunction `!.f` may be used to specify `f` as the fill:

```
   1j2 1 0j1 2 # 5 6 7 8
5 0 0 6 0 8 8
   1j2 1 0j1 2 (#!.99) 5 6 7 8
5 99 99 6 99 8 8
```

Finally, a scalar `x` is replicated to the length of `y` . This is a good way to take all items of `y` if `x` is `1`, or no items if `x` is `0` .

Boxed words to string: Monad `;:^:_1`

`;:^:_1 y` converts `y` from a list of boxed strings to a single character string with spaces between the boxed strings.

```
   ;:^:_1 ('a';'list';'of';'words')
a list of words
```

Spread: `#^:_1`

`x #^:_1 y` creates an array with the items of `y` in the positions corresponding to nonzero items of the Boolean vector `x`, and fills in the other items. `+/x` must equal `#y` .

```
   1 1 0 0 1 #^:_1 'abc'
ab  c
```

You can specify a fill atom, but if you do you must bond **x** to **#** rather than giving it as a left operand:

```
   1 1 0 0 1&#^:_1!.'x' 'abc'
abxxc
```

Choose From Lists Item-By-Item: monad `m}`

Suppose you have two arrays **a** and **b** and a Boolean list **m**, and you want to create a composite list from **a** and **b** using each item of **m** to select the corresponding item of either **a** (if the item of **m** is **0**) or **b** (if **1**). You could simply write

```
   m {"_1 a ,. b
```

and have the answer. There's nothing wrong with that, but J has a little doodad that is faster and uses less space, as long as you want to assign the result to a name. You write

```
   name =. m} a ,: b
```

(assignment with **=:** works too). This form does not create the intermediate result from dyad **,:** . If ***name*** is the same as **a** or **b**, the whole operation is done in-place.

More than two arrays may be merged this way, using the form

```
name =. m} a , b , … ,: c
```

in which each item of **m** selects from one of **a**, **b**, …, **c** . The operation is not done in-place but it avoids forming the intermediate result.

Recursion: `$:`

In tacit verbs, recursion can be performed elegantly using the verb **$:**, which stands for the longest verb-phrase it appears in (that is, the anonymous verb, created by parsing the sentence containing the **$:**, whose execution resulted in executing the **$:**). Recursion is customarily demonstrated with the factorial function, which we can write as:

```
   factorial =: (* factorial@<:) ^: (1&<)
   factorial 4
24
```

factorial(*n*) is defined as *n***factorial*(*n*-1), except that *factorial*(1) is 1. Here we just wrote out the recursion by referring to **factorial** by name. Using **$:**, we can recur without a name:

```
   (* $:@<:) ^: (1&<) 4
24
```

`$:` stands for the whole verb containing the `$:`, namely `(* $: @<:) ^: (1&<)` .

Make a Table: Adverb dyad `u/`

`x u/ y` is `x u"(lu,_) y` where `lu` is the left rank of `u` . Thus, each cell of `x` individually, and the entire `y`, are supplied as operands to `u` .

The definition is simplicity itself, and yet many J programmers stumble learning it. I think the problem comes from learning dyad `u/` by the example of a multiplication table.

The key is to note that each **cell** of `x` is applied to the entire `y` : cell, not item or atom. The rank of a cell depends on the left rank of `u` . The multiplication table comes from a verb with rank 0:

```
   1 2 3 */ 1 2 3
1 2 3
2 4 6
3 6 9
```

You can control the result by specifying the rank of `u` :

```
   (i. 2 2) ,"1/ 8 9
0 1 8 9
2 3 8 9
   (i. 2 2) ,"0 _/ 8 9
0 8 9
1 8 9

2 8 9
3 8 9
```

These results follow directly from the definition of dyad `u/` . `fndisplay` shows the details:

```
   defverbs 'comma'
   (i. 2 2) comma"1/ 8 9
+---------------+---------------+
|(0 1) comma 8 9|(2 3) comma 8 9|
+---------------+---------------+
   (i. 2 2) comma"0 _/ 8 9
+-----------+-----------+
|0 comma 8 9|1 comma 8 9|
+-----------+-----------+
|2 comma 8 9|3 comma 8 9|
+-----------+-----------+
```

Boolean Functions: Dyad `m b.`

Functions on Boolean operands

I will just illustrate Boolean dyad `m b.` by example. `m b.` is a verb with rank 0. `m`, when in the range 0-15, selects the Boolean function:

```
   9 b./~ 0 1
1 0
0 1
```

`u/~ 0 1` is the function table with `x` values running down the left and `y` values running along the top. 9 is 1001 binary (in J, `2b1001`), and the function table of `9 b.` is `1 0 0 1` if you enfile it into a vector. Similarly:

```
   , 14 b./~ 0 1
1 1 1 0
```

You can use `m b.` in place of combinations of Boolean verbs. Unfortunately, comparison verbs like `>` and `<:` have better performance than `m b.`, so you may have to pay a performance penalty if you write, for example, `2 b.` instead of `>`, even though they give the same results on Booleans:

```
   >/~ 0 1
0 0
1 0
```

J verb-equivalents for the cases of `m b.` are: 0 `0"0`; 1 `*.`; 2 `>`; 3 `["0`; 4 `<`; 5 `]"0`; 6 `~:`; 7 `+.`; 8 +`:`; 9 `=`; 10 `-.@]"0`; 11 `>:`; 12 `-.@["0`; 13 `<:`; 14 `*:`; 15 `1"0` .

Bitwise Boolean Operations on Integers

When m is in the range 16-31, dyad `m b`. specifies a ***bitwise Boolean operation*** in which the operation `(m-16) b.` is applied to corresponding bits of `x` and `y` . Since `6 b.` is exclusive OR, `22 b.` is bitwise exclusive OR:

```
   5 (22 b.) 7
2
```

The XOR operation is performed bit-by-bit.

Dyad **`32 b.`** is ***bitwise left rotate***: bits shifted off the end of the word are shifted into vacated positions at the other end.

Dyad **`33 b.`** is ***bitwise unsigned left shift***. **`x`** is the number of bits to shift **`y`** (positive **`x`** shifts left; negative **`x`** shifts right; in both cases zeros are shifted into vacated bit positions):

```
   2 (33 b.) 5
20
```

Dyad **`34 b.`** is ***bitwise signed left shift***: it differs from the unsigned shift only when **`x`** and **`y`** are both negative (i. e. right shift of a negative number), in which case the vacated bit positions are filled with 1).

If you use shift and rotate, you may need to know the word-size of your machine. One way to do that is

```
   >: 2 ^. | _1 (32 b.) 1
32
```

Operations Inside Boxes: `u L: n`, `u S: n`

`u&.>` is the recommended way to perform an operation on the contents of a box, leaving the result boxed. It is the idiom used most often by J coders and the first one to be supported by special code when performance improvements are made in the interpreter.

Sometimes your operations inside boxes require greater control than **`u&.>`** can provide. For example, you may need to operate on the innermost boxes where the boxing level varies from box to box. In these cases consider using **`u L: n`** which has infinite rank. It goes inside the operands and applies **`u`** to contents at boxing level **`n`** .

The monadic case **`u L: n y`** is the simpler one. It is defined recursively. If the boxing level of **`y`** is no more than **`n`**, the result is **`u y`** . Otherwise, **`u L: n`** is applied to each opened atom of **`y`**, and the result of that is boxed. The effect is that **`u`** is applied on each level-**`n`** subbox and the result replaces that subbox, with outer levels of boxing intact. For example,

```
   ]a =. 0;(1 2;3 4 5);<<6;7 8;9
+-+-----------+-----------+
|0|+---+-----+|+---------+|
| ||1 2|3 4 5|||+-+---+-+||
| |+---+-----+|||6|7 8|9|||
| |           ||+-+---+-+||
| |           |+---------+|
+-+-----------+-----------+
```

A boxed noun.

```
   L. a
3
```

Its boxing level is 3.

```
   # L:0 a
+-+-----+---------+
|1|+-+-+|+-------+|
| ||2|3|||+-+-+-+||
| |+-+-+|||1|2|1|||
| |     ||+-+-+-+||
| |     |+-------+|
+-+-----+---------+
```

The contents of each innermost box (where boxing level is 0) is replaced by the number of items there.

```
   # L:1 a
+-+-+---+
|1|2|+-+|
| | ||3||
| | |+-+|
+-+-+---+
```

Each level-1 boxed entity is replaced by the number of items.

```
   # L:2 a
+-+-+-+
|1|2|1|
+-+-+-+
```

Similarly for level-2 entities.

```
   # L:_2 a
+-+-+---+
|1|2|+-+|
| | ||3||
| | |+-+|
+-+-+---+
```

Negative level -**n** means ((level of **y**) minus **n**). Note that this does **not** mean '**n** levels up from the bottom of each branch of **y**'. That would result in **u**'s being applied at different levels in the different items of **`y`**; instead, the level at which **`u`** is to be applied is calculated using the level of the entire **`y`** .

The dyadic case **`x u L: n y`** is similar, but you need to know how the items of **`x`** and **`y`** correspond. During the recursion, as long as both **`x`** and **`y`** have a higher boxing level than the one specified in **`n`**, the atoms of **`x`** and **`y`** are matched as they would be matched in processing a verb with rank **`0 0`** (with replication of cells if necessary). If either operand is at the specified level, it is not changed as the items of the other operand only are opened. When both operands are at or below the specified boxing level, **`u`** is applied between them. The results of each recursion are boxed; this will give each the deeper boxing level of the two operands at each application of **`u`** .An example:

```
   (0 1;<2;3) +L:0 (10 20)
+-----+-------------+
|10 21|+-----+-----+|
|     ||12 22|13 23||
|     |+-----+-----+|
+-----+-------------+
```

`y` was passed through and applied to each level-0 entity.

```
   (0 1;<2;3) +L:0 (<<10 20)
+-------+-------------+
|+-----+|+-----+-----+|
||10 21|||12 22|13 23||
|+-----+|+-----+-----+|
+-------+-------------+
```

Once again **`y`** was applied to each entity, but because it has boxing level 2, all the results have boxing level 2.

The conjunction **`S:`** is like **`L:`**, but instead of preserving the boxing of the operands it accumulates all results into a list:

```
   (0 1;<2;3) +S:0 (<<10 20)
10 21
12 22
13 23
```

Comparison Tolerance `!.f`

Like a diamond earring that adds a sparkle to any outfit, the fit conjunction **`!.`** is a general-purpose modifier whose interpretation is up to the verb it modifies. We have seen **`!.f`** used to specify the fill atom for a verb, and to alter the formatting of monad **`":`** . Its other important use is in specifying the ***comparison tolerance*** for comparisons. A comparison like **`x = y`** calls two operands equal if they are close, where close is defined as differing by no more than the comparison tolerance times the magnitude of the larger number. If you want exact comparison, you can set the comparison tolerance to 0 using **`!.0`** :

```
   1 (=!.0) 1.000000000000001
0
   1 = 1.000000000000001
1
```

Tolerant comparison is used in the obvious places—verbs like dyad **`=`**, dyad **`>`**, and dyad **`-:`**—and also in some unobvious ones, like the verbs monad **`~.`**, monad **`~:`**, and dyad **`i.`**, and the adverb **`/.`** . For all of these you can specify comparison tolerance with **`!.f`** . You may wonder whether an exact comparison using **`!.0`** is faster than a tolerant comparison. The answer is yes, but often not by much. There is one important exception: if the comparison is used for finding equal items whose rank is greater than 0 (or are complex numbers), exact comparison can be much faster. So, if **`x`** has rank 2 or higher, it's worth the trouble to write **`x u/.!.0 y`** or **`x i.!.0 y`**; similarly use **`~.!.0 y`**, **`~:!.0 y`**, and **`x e.!.0 y`** if **`y`** has rank greater than 0.

`i.!.0` uses a completely different algorithm from dyad **`i.`** . If performance analysis shows that dyad **`i.`** is taking a lot of time, you might get an improvement by using **`i.!.0`**, even if what you are comparing is not numeric.

The **`f`** in **`!.f`** can be no larger than about **`2^_34`** . The reason for this is that there is much special code in J for handling integer operands, and for speed it assumes that comparison tolerance cannot affect integer comparisons.

The foreign **`9!:19 y`** can be used to change the default comparison tolerance, and **`9!:18 ''`** will return the current setting.

Right Shift: Monad `|.!.f`

One of my personal favorites is the infinite-rank verb monad **`|.!.f`** , defined as **`_1&(|.!.f)`**; in other words it shifts **`y`** right one place, discarding the last item and shifting an item of **`f`**s into the first position.

Generalized Transpose: Dyad `|:`

Dyad `|:` has rank `1 _` . `x |: y` rearranges `y` so that the axes given in `x` become the last axes of the result. So, if y has rank 3, `0 |: y` puts the axes of `y` into the order `1 2 3 0` and `0 2 |: y` puts them into the order `1 3 0 2` . For example:

```
   i. 2 3 4
 0  1  2  3
 4  5  6  7
 8  9 10 11

12 13 14 15
16 17 18 19
20 21 22 23
   0 |: i. 2 3 4
 0 12
 1 13
 2 14
 3 15

 4 16
 5 17
 6 18
 7 19

 8 20
 9 21
10 22
11 23
```

Formally, putting the axes into an order `p` means that `(<p{x) { p |: y` is the same as `(<x) { y` . I wish I could give you an intuitive definition but I can't.

An item of `x` can be negative to count axes from the end. The Dictionary shows how you can use boxed `x` to take elements along diagonals of `y` .

Monad `i:` and Dyad `i:`

Monad `i:` is like monad `i.`, but its interval is centered on 0 rather than starting at 0:

```
   i: 5
_5 _4 _3 _2 _1 0 1 2 3 4 5
   i: _5
5 4 3 2 1 0 _1 _2 _3 _4 _5
```

Monad `i:` can also take a complex operand to specify a different spacing between items of the result.

Dyad `i:` is like dyad `i.`, but it gives the index of the *last* occurrence (or `#x` if there is none).

Fast String Searching: `s:` (Symbols)

If you find your program taking a lot of time matching strings, you can create ***symbols*** representing the strings and then match the symbols rather than the strings themselves. The interpreter uses special code to make symbol-matching very fast.

Symbol is an atomic data type (like numeric, literal, and box). In a noun of the symbol type, each atom represents a boxed character string. You create a symbol with monad `s:` which has infinite rank. `s: y` takes an array of boxed strings `y` and creates an array of symbols of the same shape as `y` :

```
   ]sym =. s: 2 2$'abc';'def';'ghi';'jk'
`abc `def
`ghi `jk
   $sym
2 2
```

The ``'`'`` characters are a clue that `sym` is an array of symbols. The value of the top-left atom of `sym` is **not** ``'`abc'`` or `'abc'`; it is a value understood only by the interpreter. The interpreter chooses to display the text associated with the symbol, but that text is actually stored in the interpreter's private memory.

`y` in `s: y` can be a character string which is chopped into pieces using the leading character as a separator; each piece is then converted to a symbol. This is a handy way of creating a short list of symbols:

```
   s: '`abc`ghi'
`abc `ghi
```

Symbols can be operands of any verb that does not perform arithmetic; in addition, comparison between symbols is allowed with 'less than' defined to mean 'earlier in alphabetical order'.

```
   a =. s: '`abc`def`ghi`jk'
```

defines a list of 4 symbols.

```
   a i. s:<'ghi'
2
```

We create a symbol to represent `'ghi'` and find that in the list.

```
   a i. <'ghi'
4
```

Note: the boxed string `<'ghi'` is not a symbol, so it is not found in the list.

Dyad `s:` has a number of forms for operating on symbols. The only one of interest to us here is `5 s: y` which converts each symbol in `y` to its corresponding boxed string:

```
   5 s: 3 1 { a
+--+---+
|jk|def|
+--+---+
```

When a string is converted to a symbol, the interpreter allocates internal resources to hold the string's value and other information. There is no way to tell the interpreter to free the resources for a single string; this can be a problem if your symbol table is large and changes dynamically. It is possible to clear the entire symbol table (using `y=.0 s: 10` and `10 s: y`), but doing so invalidates any symbols previously created by `s: y` .

If you would like to do high-speed matching but what you want to match is not a string, consider converting to strings using `5!:5 <'y'` which converts the variable named `y` to string form.

Unicode Characters: `u:`

2-byte unicode characters can be represented by variables that have the ***unicode*** atomic data type. Such variables are created by the verb `u:` . Its use is described in the Dictionary.

Window Driver And Form Editor

Designing user interfaces is quick and painless with J's Form Editor. The Lab named Form Editor will show you how.

Tacit Programming

<< >> Contents Help

34. Tacit Programs

There is another language within J, a microcode for J as it were. Like Molière's M. Jourdain, who was astonished to find he had been speaking prose for forty years without knowing it, you have been using a small subset of this language unwittingly throughout our investigations. The hidden language describes a way of coding called ***tacit programming***, and it is time for you to learn it in full. J's tacit language is the irreducible essence of a programming language. It describes your algorithm using only the *ordering* of the J primitives you have already learned. It has a grammar without words, which you use to write programs without visible operands; yet its renouncing these seemingly essential components is self-denial rather than self-mutilation, for it retains the vigor to express any algorithm. It is as evanescent as the breeze and as powerful as the hurricane. It is a sublime creation.

We begin our exploration with this simple program:

```
   -
```

The first step toward enlightenment is to realize that something so simple *is* a program. You may object: But it can't be a program. It has no parameters. It has no name. How would I invoke it? It's a primitive, maybe. Or a verb. Or a typographical error. But not a program.

Let me make the case that it is indeed a program. I say that a bit of text deserves the title of 'program' if it produces a systematic result when executed with suitable arguments. And I say that the program `'-'` satisfies this definition. Certainly I can supply it with arguments

```
   5 7 - 2
3 5
```

and get the expected result. The program does not refer to its operands explicitly, but as long as we make the agreement that the arguments to a program appear as nouns to its left and right, the program has no doubt about how to find its arguments. Of course, we have been using this convention for verbs all along.

I can give this program a name:

```
   minus =: -
   5 7 minus 2
3 5
```

`minus` can be used in place of `-` anywhere. When we look at the value of `minus`, we see what it contains:

```
   9!:3 (5)  NB. Do this once to select simplified
display

   minus
-
```

`minus` is equivalent to `-` . In J we call it a verb, but it has all the essential features of a program. In fact, `minus` is *two* programs, because it can be executed monadically as well:

```
   minus 6 8
_6 _8
```

The point is that every J verb, even the primitives and the compound verbs, is a program in the usual sense. Verbs like `minus`, that do not mention their operands by name but instead apply them according to J's parsing rules, are called ***tacit verbs***. Verbs created by `m :n`, like `dyad : '+/ y.'`, that mention their operands by name are called ***explicit verbs***. The compound verbs we have learned already are examples of tacit verbs. Some of the verbs that we have had occasion to define so far can be written in tacit form:

```
addrow =: monad : '+/ y.'"1
```

could be rewritten as

```
addrow =: +/"1
```

and

```
v =: dyad : '1.04 * x. + y.'
```

is equivalent to

```
v =: 1.04&*@:+
```

as we have seen. We have already encountered tacit definitions without noticing:

```
   dotprod =: +/@:*"1
   1 2 3 dotprod 1 2 3
14
```

and we can certainly define more.

```
   sortup =: /:~
```

defines a verb that sorts its operand into ascending order:

```
   sortup 31 41 59 26 53
26 31 41 53 59
```

Some of the verbs we have encountered seem too complex for a compound verb. For example, in

```
   mean =: monad : '(+/ y.) % #y.'
```

we need to perform two operations on **`y.`** and combine their results, something that is beyond the capabilities of the compound verbs we have encountered so far. We will next learn how to produce a tacit equivalent for **`mean`** and a great many other verbs.

<<   >>   Contents   Help

35. First Look At Forks

Before we learn the rules for making tacit forms, you should understand why you are going to the trouble of learning how to write programs that hide their operands. First, they are extraordinarily compact. The explicit definitions we have written so far are laconically terse by the standards of most computer languages, but they will seem positively windy compared to the tacit forms. Second, the shorter definitions are easier to combine with other verbs, and with the modifiers that add so much power to J expressions. Third, the tacit definitions are parsed when they are defined, in contrast to explicit definitions, in which each line is parsed as it is executed; we reduce interpretive overhead by using tacit forms. Fourth, in learning to write tacit verbs you are also learning to write tacit adverbs and conjunctions, with which you will be able to craft your own private toolkit of modifiers that you can use to combine verbs in ways that are useful to your application.

In what follows, **N*x*** will represent a noun, **V*x*** a verb, **C*x*** a conjunction, and **A*x*** an adverb, where ***x*** is any suffix.

We begin by observing that the rules we have learned so far give no meaning to some combinations of words. Consider three verbs in a row, with no noun to operate on, as in the sequence

```
   (V0 V1 V2)
```

where each **V*n*** represents a verb—an example would be `((+/) % #)` . Without some special rules, we have no way to interpret this sequence. Such sequences of words that cannot immediately be executed to produce a result are called ***trains***. Examples are `C0 C1 A2`, `V0 V1`, and the `V0 V1 V2` we are considering now.

Understanding tacit programming will largely be a matter of understanding how trains are parsed and executed. You will learn that `(V0 V1 V2)` is a new verb that can be applied to noun operands, and you will learn how it applies to nouns. To begin with, observe that there is no reason that `(V0 V1 V2) N` should be the same as `V0 V1 V2 N` which as we know is `(V0 (V1 (V2 N)))` .

The meaning J assigns to `(V0 V1 V2) Ny` is:

```
   (V0 V1 V2) Ny  is  (V0 Ny) V1 (V2 Ny)
```

This substitution goes by the name ***monadic fork***. I think finding this definition was a stroke of brilliance in the design of J. An example of the use of the fork is:

```
   (+/ % #) 4 6 8
6
```

which calculates the mean of the operand. It is processed using the substitution rule above as

```
   (+/ 4 6 8) % (# 4 6 8)
6
```

which divides the sum of the items in the list by the number of items in the list. You can use `fndisplay` to help yourself see how the substitutions are made:

```
   defverbs 'plus"0 div"0 tally'
   (plus/ div tally) 4 6 8
+---------------------------------+
|(4 plus 6 plus 8) div tally 4 6 8|
+---------------------------------+
```

The sequence `(+/ % #)` is a verb. It can therefore be assigned to a name:

```
   mean =: (+/ % #)
```

or

```
   mean =: +/ % #
```

and then used by that name:

```
   mean 12 18 24
18
```

Neat, eh? With just 4 symbols we described a program to take the mean of a list of numbers (or a list of lists…). The beauty and the power are in the way the operands and verbs are connected; that's what we'll be learning in the next few chapters.

At this point you may be impressed with the economy of the monadic fork but a bit confused about the details. For example, we said that `(V0 V1 V2) Ny` is not the same as `V0 V1 V2 Ny` and yet we said that `mean =: (+/ % #)` is the same as `mean =: +/ % #` . How can that be? If we use the version without parentheses, why doesn't `mean 12 18 24` get evaluated like

```
   +/ % # 12 18 24
0.333333
```

?

I could give you a simple rule of thumb, namely that you can always imagine an extra set of parentheses around any value assigned to a name. That would be true but misleading, because it would encourage you to think that the values of defined names are substituted into a sentence before the sentence is executed. That gets it backwards: in reality **the operands are supplied to the stored definitions**. In fact, the execution of a J sentence is a subtle alternation between creating definitions and executing them. We will take the next couple of chapters to give you a thorough understanding of execution, after which we will return to see what magic we can work with forks and their brethren.

<<   >>   Contents   Help

36. Parsing and Execution I

I hope your hunger for understanding will be enough to motivate you to read a couple of difficult chapters. If you do, you will learn something few J programmers know—what really happens when J executes a sentence. In this chapter we will analyze sentences from the top down, to get an idea for the order of execution. In the next chapter we will follow the interpreter as it alternately parses and executes sentences from the bottom up.

Since the understanding of parsing and execution that you have developed during your work so far is probably a bit inaccurate, we will work through examples of increasing complexity.

```
   9!:3 (5)  NB. Do this once to select simplified
display

   &.
&.
```

With only one word, there are no operands and nothing to execute, so the result of the sentence is the word itself: the conjunction `&.` .

```
   -:&.^.
-:&.^.
```

The result of executing `-:&.^.`, i. e. executing `&.` with `-:` and `^.` as operands, is an ***anonymous verb***. This anonymous verb will execute according to the definition of `&.`, given its operands `-:` and `^.` (i. e. `-:&.^. y` will be `^ -: ^. y`).

Note that the conjunction `&.` is executed without reference to the operand of the anonymous verb (indeed, in this case there is no such operand and the anonymous verb is the result of the sentence). We could assign the anonymous verb to a name, in which case it would no longer be anonymous (e. g. `sqrt =: -:&.^.`); without such an assignment we will refer to it here by the nickname ***`av`*** . The value of ***`av`*** is the verb described by `-:&.^.` .

```
   -:&.^. 16
4
```

We know that this is executed as `^ -: ^. 16`; let's see how that happens.

Conjunctions are executed before verbs, so first `-:&.^.` will be executed to produce the anonymous verb we called ***av*** . Then ***av*** is executed with the operand `16` . ***av*** operates according to the definition of `&.` : it produces the same result as `^ -: ^. 16` (but it may use a different algorithm than executing `^ -: ^. 16` directly).

It appears that `&.` was executed twice: once to create ***av*** and then again during the execution of ***av*** . No, it was executed only once, to create ***av*** . ***av*** operates according to the definition of `&.`, but it is ***av*** that is executing, not `&.` . The confusion arises because of the way the interpreter displays ***av*** . There is no better way to show a verb that performs the function `-:&.^.` than to show the way the verb was created, i. e. with the characters '`-:&.^.`', but you should think of this as an exhibition of the pedigree of ***av***, and an assurance of its good behavior, rather than a list of functions to be executed. In fact, part of the reason for J's good performance comes from its recognizing functions that can be combined efficiently and providing customized routines to handle anonymous verbs that call for those combinations.

Confusion between a conjunction and the anonymous verb it produces is most likely when the conjunction is one you wrote using `conjunction define` or `2 : n` . In most cases the text of the conjunction actually describes a derived verb, and it is natural for you to say 'the conjunction `C` is executed with operands `u.`, `v.`, and `y.`' rather than the more accurate 'the anonymous verb created by the application of `C` to `u` and `v` is executed, with `u.` and `v.` available during the interpretation of the text of `C` and with `y.` as the operand'. Such confusion is almost always harmless, but let's go through a few examples so you can see the layers of execution:

```
   2 : 'u.'
2 : 'u.'
```

We execute `2 : 'u.'` and the result is an anonymous conjunction that we'll call ***ac1*** . The display of ***ac1*** shows where it came from. When ***ac1*** is executed, its result will be its left operand.

```
   +: (2 : 'u.') -:
+:
```

Here `2 : 'u.'` is executed first to produce ***ac1***; then ***ac1*** is executed with left operand of `+:` and right operand of `-:` . The result is an anonymous verb that we'll call ***av1***; its value is the verb `+:` which was the left operand to ***ac1*** .

```
   +: (2 : 'u.') -:  5
10
```

Remember, `(2 : 'u.')` is a conjunction (the conjunction we have called ***ac1***), and conjunctions are executed before verbs, so this is executed as `(+: (2 : 'u.') -:) 5`, which becomes ***av1*** `5` . We execute ***av1*** with the operand `5` . Monad `+:` doubles its operand, and the result is `10` .

We know that a conjunction can produce a conjunction result. That's how explicit conjunctions are formed: executing the conjunction `:` with left operand `2`, as in `2 : n`, produces a conjunction. There is nothing special about `2 :n` : any conjunction is allowed to produce a conjunction result:

```
   2 : '&'
2 : (,'&')
```

We execute `2 : '&'` and the result is an anonymous conjunction that we'll call ***ac2*** . The display of ***ac2*** shows where it came from. (the `,` in the display of ***ac2*** is harmless, a reminder that internally the anonymous entity resulting from `m :n` stores `n` as a list of characters.)

```
   +: (2 : '&') -:
&
```

We execute ***ac2*** with left operand of `+:` and right operand of `-:` . The result is an anonymous conjunction that we'll call ***ac3*** . ***ac3*** is a conjunction because its value `&` (the last sentence executed by ***ac2***) is a conjunction. Yes, `&` by itself can be a result: modifiers can return any primary part of speech (but try to return a conjunction from a verb and you will get an error).

Note that this is **not** the same as `u.&v.` : that would also be a valid return value from a conjunction, but `u.` and `v.` would be substituted and `&` would be executed to make the returned value an anonymous verb with the description `u&v` .

Make sure you see why the `+:` and `-:` disappeared. First, the conjunction `:` was executed with operands `2` and `'&'`; that produced a conjunction ***ac2*** which was then executed with operands `+:` and `-:`; but the defining text of ***ac2*** does not look at its operands; it simply produces the value `&` . So, the operands to ***ac2*** disappear without a trace, and the result of the whole phrase is a conjunction with the value `&` .

```
   2 (+: (2 : '&') -:) *
2&*
```

Continuing the example, we execute ***ac3*** (which was just the conjunction `&`) with left operand `2` and right operand `*` . The result is the anonymous verb ***av2*** which will execute as `2&*` .

```
   2 (+: (2 : '&') -:) *   5
10
```

Finally, we execute ***av2*** with the operand **5**, and get the result **10** .

Explicit modifiers that refer to the **operands** of their derived verb (as **x.** or **y.**) come in for special treatment. A simple example is the conjunction defined by

```
   2 : 'u. v. y.'
2 : 'u. v. y.'
```

We execute **2 : 'u. v. y.'** and the result is an anonymous conjunction that we'll call ***ac4*** . You can't tell it from the display, but ***ac4*** is a special kind of conjunction. Because it refers to **y.**, the *text* of ***ac4*** can be executed only as a verb (only then are **x.** and **y.** meaningful). The stored ***ac4*** makes note of this fact.

```
   +: (2 : 'u. v. y.') -   5
_10
```

When ***ac4*** itself is executed (as **+: (2 : 'u. v. y.') -** here—since ***ac4*** is a conjunction it is executed before its result is applied to the noun operand **5**), the text of ***ac4*** is not interpreted (as it was in our other examples). Instead, the new anonymous verb ***av3*** is created. ***av3*** contains the defining text of ***ac4***, along with the operands that were given to ***ac4*** (**+:** and **-** here). When the verb ***av3*** is executed as in the line above, the text of ***ac4*** is finally interpreted, with the operands of ***ac4*** (**+:** and **-** here) available as **u.** and **v.**, and the noun operands of ***av3*** (**5** here) available as **y.** (and **x.** if the invocation is dyadic); the result is the result of **+: - 5** .

<< >> Contents Help

37. Parsing and Execution II

Now that you understand what an anonymous verb/adverb/conjunction is, you are ready to follow parsing and execution word by word. We will finally abandon all shortcuts and process sentences exactly as the interpreter does.

In any compiled language, a program is broken into words (**tokens**) and then parsed, and code is generated from the parsed result. Not so in J: a sentence is broken into words, but the sentence is not fully parsed; rather, parsing and execution proceed simultaneously, scanning the text from right to left. ***Parsing*** finds patterns in the sentence that are then executed. ***Execution*** includes the usual supplying of noun operands to verbs to obtain a result from the verb, but also other actions: supplying verb and noun operands to conjunctions and adverbs to produce derived entities, and recognition of other sequences that we will learn soon. Execution of a bit of a sentence, which we will call a ***fragment***, consists of replacing the fragment with an appropriate *single word*, namely the result of executing the fragment. In the simple case, where the fragment is the invocation of a verb (i. e. the fragment looks like ***`verb noun`*** or ***`noun verb noun`***), the word that replaces it is the noun that is the result of the verb. If the fragment is the invocation of a modifier, the result of executing it will be a noun or a derived verb/adverb/conjunction. A noun is nothing but its value, but the derived verb/adverb/conjunction will itself eventually be executed: it is called an ***anonymous verb/adverb/conjunction*** and is saved by the interpreter in a private form, and the single word used to replace the fragment is a reference to this anonymous verb/adverb/conjunction (for the case of an anonymous verb, you may think of the single word as a **pointer** to a **function** that performs according to the definition of the anonymous verb). In all cases the word replacing the fragment has a definite part of speech, and if it is a verb, a definite rank.

The Parsing Table

Execution of a sentence begins by breaking the sentence into words. The words (with a beginning-of-line marker, shown here as §, prepended) become the initial contents of the ***unprocessed word list***. A push-down ***stack*** will also be used during execution; it is initially empty. Execution of the sentence is performed by repetition of the ***parsing step*** which comprises: (1) examining the top 4 elements of the stack to see if they match one of the 10 executable patterns; (2) if a match was found, executing the executable portion of the stack (what we called the executable fragment in the last chapter), resulting in a single word which replaces the fragment on the stack; (3) if no match was found, moving the rightmost word of the unprocessed word list into the leftmost position of the stack, pushing the rest of the stack to the right. Execution finishes when there are no unprocessed words and the stack does not contain an executable pattern. Note that execution of a fragment may leave the stack matching one of the executable patterns, so several sequential parsing steps may perform an execution without moving anything onto the stack. After all words have been processed, the stack should contain a beginning-of-line marker followed by a single word which becomes the result of the sentence.

To follow the parsing we need to know what patterns at the top of the stack contain an executable fragment. The ***parsing table*** below gives the complete list. More than one symbol in a box means that any one of them matches the box. **name** means any valid variable name, and C, A, V, and N stand for conjunction, adverb, verb, and noun respectively.

leftmost stack word	**other stack words**			**action**
§ =. =: (	**V**	**N**	anything	0 Monad
§ =. =: (A V N	V	**V**	**N**	1 Monad
§ =. =: (A V N	**N**	**V**	**N**	2 Dyad
§ =. =: (A V N	**V N**	**A**	anything	3 Adverb
§ =. =: (A V N	**V N**	**C**	**V N**	4 Conj
§ =. =: (A V N	**V**	**V**	**V**	5 Fork
§ =. =: (	**C A V N**	**C A V N**	anything	6 Hook/Adverb
name N	**=. =:**	**C A V N**	anything	7 Is
(	**C A V N**	**)**	anything	8 Paren

The lines in the parsing table are processed in order. If the leftmost 4 words on the stack match a line in the table, the fragment (those words on the stack which are in boldface in the parsing table) is executed and replaced on the stack by the single word returned. Because the fragment is always either two or three words long, it is officially known as a ***bident*** or ***trident***. The last column of the parsing table gives a description of what execution of the fragment entails.

You will have an easier time following the parsing if you note that the leftmost word in the executable pattern is usually one of **§ =. =: (** ***A V N*** . This means that you can scan from the right until you hit a word that matches one of those before you even start checking for an executable pattern. If you find one of **§ =. =: (** ***A V N*** and it doesn't match an executable pattern, keep looking for the next occurrence.

Note that the leftmost stack word in the parsing table is never a conjunction. This is the ultimate source of our long-noted rule that conjunctions associate left-to-right: a conjunction can be executed when it appears in the third stack position, but if another conjunction is in the leftmost position then, the stack will always be pushed down to examine that conjunction's left argument.

Examples Of Parsing And Execution

We will now follow a few sentences through parsing. We will represent anonymous entities by names in italics, with an indication of how the anonymous entity was created. Up till now in this book we have scarcely noticed that the term 'verb' was used both for an entity that can be applied to a noun to produce a noun result, and also for the *name* of that entity. This ambiguity will continue—being precise would be too cumbersome—but you should be aware of it. When we say 'the result is *av*, defined as +/', that means that an anonymous verb was created whose function is described as `+/`, and the nickname we are giving it—the word, that is, that goes on the execution stack to betoken this verb—is *av*.

Sentence: `+/2*a` where `a` is `1 2 3`

unprocessed word list	stack	line
§ + / 2 * a		
§ + / 2 *	1 2 3 (not executable)	
§ + / 2	* 1 2 3 (not executable)	
§ + /	2 * 1 2 3 (not executable)	
§ +	/ **2 * 1 2 3** (result 2 4 6)	2
	§ + / 2 4 6 (result *av*, defined as +/)	3
	§ ***av* 2 4 6** (result 12)	0
	§ 12	

The column labeled 'line' indicates which line of the parsing table was matched by the stack. The fragment is marked in boldface and underlined. Note that when the noun `a` moved onto the stack, its *value* was moved; when a named verb, adverb, or conjunction is moved onto the stack, only the *name* is moved. Note also that the noun's value (`1 2 3` here) is a single word.

From now on we will omit the lines that do not contain an executable fragment.

Sentence: `mean =: +/ % #`

unprocessed word list	stack	line
§ mean =: + / % #		
§ mean	=: + / % # (result *av1*, defined as +/)	3
§ mean	=. ***av1* % #** (result *av2*, defined as *av1* % #)	5

	§ **mean =: av2** (result *av2*; `mean` is assigned *av2*)	7
	§ *av2*	

I want to emphasize that what is assigned to `mean` is the *result of parsing* `+/ % #` . It is **not** the sequence `+/ % #`, but rather a single verb which performs the function described by the fork. Now you see why putting parentheses around the definition doesn't matter: *av2* would be parsed the same either way.

unprocessed word list	stack	line
§ mean 4 5 6		
	§ **mean 4 5 6** (result 5)	0
	§ 5	

Sentence: `mean 4 5 6`

Since `mean` is the result from parsing `+/ % #`, it is executed without further ado. As you can see, a single 'execution' step can trigger a cascade of processing as each verb referred to by an executing verb is executed in turn. Here, execution of `mean` does the entire processing of the fork, returning the result `5` . The verb to be executed can be quite complex, and can have a mixture of named and anonymous components, as in the next example.

Sentence: `(mean - (+/ % #)&.(^."1)) 4 5 6` (find the difference between arithmetic and geometric mean)

unprocessed word list	stack	line
§ (mean - (+ / % #) &. (^. " 1)) 4 5 6		
§ (mean - (+ / % #) &.	(**^. " 1**)) 4 5 6 (result *av1*, defined as ^. " 1)	4
§ (mean - (+ / % #) &.	**(*av1*)**) 4 5 6 (result *av1*)	8
§ (mean -	(+ / % #) &. *av1*) 4 5 6 (result *av2*, defined as +/	3
§ (mean -	(***av2* % #**) &. *av1*) 4 5 6 (result *av3*, defined as *av2* % #)	5
§ (mean -	**(*av3*)** &. *av1*) 4 5 6 (result *av3*)	8
§ (mean	- ***av3* &. *av1***) 4 5 6 (result *av4*, defined as *av3* &. *av1*)	4

	§(**mean - *av4***) 4 5 6 (result av5, defined as mean - *av4*)	5
	§**(*av5*)** 4 5 6 (result *av5*)	8
	§ ***av5* 4 5 6** (result **`0.0675759`**)	0
	§ 0.0675759	

Again, there was only one execution of a verb. It happened at the very end: after *av5* was created, it was executed, and its execution included the execution of everything else.

Sentence: **`inc =: ({.a)&+`** where **`a`** is **`4 5 6`**

unprocessed word list	stack	line
§ inc =: ({. a) & +		
§ inc =:	(**{. 4 5 6**) & + (result **`4`**)	0
§ inc =:	**(4)** & + (result **`4`**)	8
§ inc	=: **4 & +** (result *av*, defined as 4&+)	4
§	**inc =: *av*** (result *av*; **`inc`** is assigned *av*)	7
	§ *av*	

This illustrates an important point. Even in the middle of a complex definition, verbs are applied to nouns wherever possible. And, the value of a noun in a definition is the value at the time the definition was *parsed*. If a parsed definition refers to a verb, it does so by name, so the value of a verb is its value when it is *executed*.

The remaining examples are curiosities to show you that it's worth your trouble to learn the intricacies of parsing.

Sentence: **`a + a =. 5`**

unprocessed word list	stack	line
§ a + a =. 5		
§ a +	**a =. 5** (result is **`5`**; **`a`** is assigned **`5`**)	0
	§ **5 + 5** (result is **`10`**)	2
	§ 10	

`a` is assigned a value just before that value is pushed onto the stack.

Sentence: **`2 +: (2 : '&') -: *   5`**

unprocessed word list	stack	line
§ 2 +: (2 : '&') -: * 5		
§ 2 +:	(**2 : '&'**) -: * 5 (result is *ac1*, defined as 2 : '&')	4
§ 2 +:	**(*ac1*)** -: * 5 (result is *ac1*)	8
§	2 **+: *ac1* -:** * 5 (result is *ac2*, defined as &)	4
	§ **2 *ac2*** * 5 (result is *av*, defined as 2&*)	4
	§ ***av* 5** (result is **10**)	0
	§ 10	

Look at what happens when we omit the parentheses:

Sentence: **`2 +: 2 : '&' -: *   5`**

unprocessed word list	stack	line
§ 2 +: 2 : '&' -: * 5		
§ 2 +: 2 :	'&' -: * 5 (result is **1**)	1
§ 2	+: **2 : '&'** -: 1 (result is *ac1*, defined as 2 : '&')	4
§	2 **+: *ac1* -:** 1 (result is *ac2*, defined as &)	4
	§ **2 *ac2* 1** (domain error: 2&1 is illegal)	4

The omission produces a subtle but fatal change in the parsing order. As the Dictionary says, "it may be necessary to parenthesize an adverbial or conjunctival phrase that produces anything other than a noun or verb". Now you see why.

Undefined Words

If you try to execute a nonexistent verb, you get an error:

```
   z 5
|value error: z
|       z 5
```

However, that error occurs during execution of the name, not during its parsing. During parsing, an undefined name is assumed to be a verb of infinite rank. This allows you to write verbs that refer to each other, and relieves you from having to be scrupulous about the order in which you define verbs. For example:

```
   a =: z
```

This produces a verb **`a`** which, when executed, will execute **`z`**.

```
   z =: +
   a 5
5
```

With **`z`** defined, **`a`** executes correctly. Of course, it's OK to assign **`z`** to another verb too:

```
   b =: z
   b 5
5
```

Now, can you tell me what **`+/@b 1 2 3`** will do? Take a minute to figure it out (Hint: note that I used **`@`** rather than **`@:`**).

```
   +/@b 1 2 3
1 2 3
```

Because **`b`** has rank 0, **`+/@b`** also has rank zero, so the summing is applied to atoms individually and we get a list result. Do you think **`+/@a 1 2 3`** will have the same result?

```
   +/@a 1 2 3
6
```

Even though **`a`** has the same *value* as **`b`**, its *rank* is different. **`a`**'s rank was assigned when it was parsed, and at that time **`z`** was assumed to have infinite rank. **`b`**'s rank was assigned when it was parsed too, but by that time **`z`** had been defined with rank 0. You can win a lot of bar bets with this one if you hang out with the right crowd.

<<   >>   Contents   Help

38. Forks, Hooks, and Compound Adverbs

Now that you understand execution, and in particular how anonymous entities are created and then executed, you are ready to see forks used in some practical applications. This will be a relief after the last two chapters of theory.

You have learned the rule for the trident called the monadic fork:

```
   (V0 V1 V2) Ny  is  (V0 Ny) V1 (V2 Ny)
```

Now learn the other 3 bidents/tridents involving only verbs. The ***dyadic fork***:

```
   Nx (V0 V1 V2) Ny  is  (Nx V0 Ny) V1 (Nx V2 Ny)
```

The ***monadic hook***:

```
   (V0 V1) Ny  is  Ny V0 (V1 Ny)
```

The ***dyadic hook***:

```
   Nx (V0 V1) Ny  is  Nx V0 (V1 Ny)
```

The purpose of taking you through the 2 preceding chapters was for you to understand that 'is' in these definitions is shorthand for 'replaces the parenthesized part with an anonymous entity that when executed on an `x` and `y` produces the same result as' (you don't have to be a politician to hesitate over the meaning of 'is').

It may be helpful to think of these bidents and tridents as ghostly conjunctions, with no actual symbol, that create an entity (the bident/trident) out of the sequence of verbs. The entity so created is quite real: it is executed just like any anonymous verb created by a modifier.

You can see that for both the hooks and the forks, the monadic case is derived from the dyadic: for forks by omitting `Nx`, and for hooks by replacing `Nx` with `Ny` . The verbs produced by hooks and forks have infinite rank.

Using hooks and forks, assisted by all the modifiers we have learned, we can produce any function of 2 operands. If we have more than 2 operands, we can link them together into a boxed list using dyad `;` and then extract the pieces as needed using dyad `{::` . For the rest of this chapter we will show examples of functions turned into tacit verbs using hooks and forks. If I don't show the expansion using the bident/trident rules, you should produce it yourself.

To find how much **x** has changed from **y**, as a percentage of **y** :

```
   pctchg =: 100&*@:(- % ])
   12 pctchg 10
20
```

This becomes **100 * (x - y) % y** . Note the use of **]** to select the original **y** operand. Similarly, **[** can be used to select the original **x** operand. Tacit verbs make heavy use of **[** and **]** .

Another way to code **pctchg** is

```
   pctchg =: 100"_ * (- % ])
```

where we used the constant verb **100"_** which produces **100** no matter what its operands are. Which of these forms you prefer is a matter of taste.

fndisplay is very helpful in understanding tacit verbs. The two versions of **pctchg** are displayed as

```
   defverbs 'times"0 minus"0 div"0'
   defnouns 'x y'
   x 100&times@:(minus div ]) y
+---------------------------+
|100 times (x minus y) div y|
+---------------------------+
   x (100"_ times (minus div ])) y
+---------------------------+
|100 times (x minus y) div y|
+---------------------------+
```

I encourage you to use **fndisplay** to expand any tacit definitions that are troublesome.

To find the elements common to **x** and **y**, keeping the same order as in **x** :

```
   setintersect =: e. # [
   3 1 4 1 5 9 setintersect 4 6 9
4 9
```

This becomes **(x e. y) # x** . You can see that the identity verbs **[** and **]** are useful for steering operands through hooks and forks. As an exercise, see how the alternative version **([-. -.)** produces the same result.

To list all the indexes of the 1s in a Boolean list:

```
   booltondx =: (# i.@:#)"1
   booltondx 0 1 0 1 0 0 1
1 3 6
```

Note that we are careful to give our verb a rank of 1, since it works only with lists. The primitive `I.` has the same effect.

To find the difference between the largest and smallest items in a list:

```
   range =: (>./ - <./)"1
   range 3 1 4 1 5 9
8
```

To find the index of the largest item in a list:

```
   indexmax =: (i. >./)"1
   indexmax 3 1 4 1 5 9 2 6 5 3 5
5
```

To create a Boolean list with a `1` at each position that is different from the previous position:

```
   changeflag =: 1: , 2: ~:/\ ]
   changeflag 1 1 2 2 7 7 7 3 3 4 5 8 8
1 0 1 0 1 0 0 1 0 1 1 1 0
```

We could have done this without using forks, with `(1&,)@:(2&(~:/\)` . Which version you use is a matter of taste. The number verbs `_9:` through `9:` are very useful if you like forks rather than long conjunction chains. Note that `changeflag` is executed as if parenthesized `(1: , (2: (~:/\) ]))` . If you work with long trains of verbs like this you will soon notice that if you count the verbs from the right (starting at 0, of course), the even-numbered verbs have the original `x` and `y` applied, and the odd-numbered ones combine the results from the even-numbered.

To replace multiple spaces by a single space:

```
   delmb   =: ((#~ -.)  '  '&E.)"1
   delmb 'abc   nb'
abc nb
```

To create an array in which each item is a list of (value, number of times that value appeared):

```
histogram =: ~. ,. #/.~
   histogram 3 1 4 1 5 9 2 6 5 3 5 8 9 7 9
3 2
1 2
4 1
5 3
```

```
9 3
2 1
6 1
8 1
7 1
```

To append the contents of the first item of `x` in front of `y` and the contents of the last item of `x` behind `y` :

```
    enclose =: >@:{.@:[ , ] , >@:{:@[
   '*' enclose 'xyz'
*xyz*
   '()' enclose 'abc'
(abc)
```

To produce `1` if all the items of `y` are equal, `0` if not

```
   allitemsequal =: -: 1&|.
   allitemsequal 1 1 1 1 1
1
   allitemsequal 1 1 1 2 1
0
```

To extend `x` to the length of `y`, where the items of `y` supply default values for the corresponding omitted items of `x` :

```
   default =: [ , (}.~ #)~
   ('abc';2) default 'Defname';0;'Deftype';100
+---+-+-------+---+
|abc|2|Deftype|100|
+---+-+-------+---+
   ('abc';2;'xyz';30) default 'Defname';0;'Deftype';100
+---+-+---+--+
|abc|2|xyz|30|
+---+-+---+--+
```

The verb `[:`, which we met as a way to cause an error, has a special meaning in a fork. As the leftmost verb of the fork, `[:` means 'ignore the left branch'. So, `Nx ([: V1 V2) Ny` is `V1 Nx V2 Ny` and `([: V1 V2) Ny` is `V1 V2 Ny` . In both cases, `([: V1 V2)` is equivalent to `V1@:V2` . Almost always, the choice between one form or the other is a matter of taste. Do not fear that the extra word in the fork leads to slower execution; the `[:` is not executed—it is recognized by the parser when it creates the anonymous verb for the fork.

As a final example, here is a definition of the word-counting example we wrote earlier. See if you can convince yourself that it is equivalent to **`wc2`** :

```
WS =: ' ',TAB,LF
wc3 =: (# , (*. -.@(|.!:0))@(e.&WS) , +/@(LF&=))
@ReadFile
```

I could go on with pages more of hooks and forks for you to study, but what you really need is to write some yourself. It will be a frustrating experience for the first few weeks as you struggle to jigsaw your verbs into pieces that have one or two operands and yet fit together to perform a complex function. It's not a necessary skill—you can get along acceptably in J writing mostly explicit definitions, with an occasional fork thrown in where obvious—but it is a useful, honorable, and satisfying one. Learning to write tacit verbs is like learning to walk with a book balanced on your head: it will slow you down at first, but in the end you'll stand taller for it.

The book *J Phrases*, which is part of the J release, has dozens of interesting examples of tacit programs which you can use as exercises.

Tacit and Compound Adverbs

Adverbs as well as verbs can be defined tacitly. Any sequence of adverbs is also an adverb, and applies the component adverbs one by one to its left operand. For example,

```
   onprefixes =: /\
```

defines an adverb that applies **`/`** followed by **`\`**, as can be seen by

```
   + onprefixes 1 2 3 4
1 3 6 10
```

A conjunction with one operand also defines an adverb. A conjunction needs two operands, but if you supply one, the combination is treated as an adverb that attaches its operand to the empty side of the conjunction. For example, **`(2&)`** is an adverb, and **`+ (2&)`** is equivalent to **`2&+`** . For another example, the J startup scripts define

```
   each =: &.>
```

so that

```
   >: each 1 2 3
```

is equivalent to

```
   >:&.> 1 2 3
```

Referring To a Noun In a Tacit Verb

Suppose you want a verb `v` that returns the current value of the noun `n` (maybe it's a button handler). Suppose you want `v` to be tacitly defined. How would you do it? You can't use

```
v =: n
```

because that would use the value of `n` at the time `v` is *defined* (in fact, `v` would be a noun), and you want the value of `n` when `v` is *executed*. Use this trick:

```
v =: ".@:('n'"_)
```

When this is executed, the operand is ignored and replaced by the string `'n'`, which is then executed by `".` to produce the value of the noun `n` .

39. Readable Tacit Definitions

Heron's formula for the area of a triangle, given the lengths of the sides *a*, *b*, and *c*, is *sqrt*(*s**(*s*-*a*)*(*s*-*b*)**s*-*c*)) where *s* is (*a*+*b*+*c*)%2 . This can be written

```
   triarea =: [: %: [: */ -:@:(+/) - 0: , ]
   triarea 3 4 5
6
```

Regardless of your diligence in commenting your code and the level of J expertise in you and the sorry wretches who will have to read it, long tacit definitions like this can become trackless wastelands, Saharas of verb following verb unendingly with nothing to suggest an interpretation of the intermediate results. I know of two ways to mark a trail through such a definition. The first is to develop a regimen for using spaces and parentheses to help the reader group the parts. I would write

```
   triarea =: [: %: [: */  (-:@:(+/))   -   0:,]
```

The second way is to split the line into multiple lines, where each line can have a comment or a verb-name indicating what it produces. This approach, carried almost to an extreme, would yield

```
   semiperimeter =: -:@:(+/)
   factors =: semiperimeter - 0:,]
   product =: [: */ factors
   triarea =: [: %: product
```

Combining the two approaches, you can find a comfortable level of complexity.

Flatten a Verb: Adverb `f.`

Splitting the definition into many lines has the unfortunate side-effect that all the names referred to by `triarea` must be defined when `triarea` is executed:

```
   triarea
[: %: product
```

`triarea` refers to `product` which refers to `factors` which refers to `semiperimeter` . If you define many tacit verbs this way, the result is pollution of the namespace. To leave a smaller footprint, use *private* assignment for all the names except the name that will be public, and use the adverb `f.` which replaces names in its operand with their definitions:

```
   semiperimeter =. -:@:(+/)
   factors =. semiperimeter - 0:,]
   product =. [: */ factors
   triarea =: ([: %: product) f.
   triarea
 [: %: [: */ -:@:(+/) - 0: , ]
```

If these verbs are run from a script, the temporary verbs will disappear (since they were assigned by private assignment), leaving only the main verb `triarea` .

`f.` is also used in initialization of objects, as you can learn in the Lab for Object-Oriented Programming.

Note that `f.` has no effect on explicit definitions.

Using `f.` to improve performance

Flattening a verb has two beneficial effects on performance. The first is easy to see by comparing two equivalent sentences:

```
   a =. i. 100000
   abs0 =: 3 : '| y.' "0
   6!:2 'abs0 a'
3.17136
   abs1 =: 3 : '| y.'
   6!:2 'abs1"0 a'
3.54908
```

To be sure, in each case we have committed the crime of applying an explicitly-defined verb at a low rank (`|"0 a` executes in time `0.006`), but that is not the point. Why is `abs1"0` slower than `abs0`? Each one reinterprets its verb for each atom of `a` .

The answer is that when `abs1"0` is executed, the definition of `abs1` must be looked up for every atom of `a` (for all the interpreter knows, `abs1` might be redefined during its execution). The time spent doing this lookup accounts for the difference in time between `abs0` and `abs1"0` . If we eliminate the lookup of the name `abs1`, that time is saved:

```
   6!:2 'abs1 f."0 a'
3.00941
```

The important lesson to learn from this is that you should define your verbs with the proper rank. That will eliminate superfluous name lookups. In exceptional cases you may use `f.` to avoid name lookups during execution of a complex verb.

The second case where `f.` can improve performance is useful only for those users who feel compelled to redefine the J primitives with mnemonic names. This is a practice that I strongly deprecate, and if you don't heed my advice, the interpreter stands ready to punish you. See the disaster that can result when the primitives are replaced by mnemonic names:

```
   tally =: #
   a =. 100000 $ i. 6
   b =. i. 100000 10
   6!:2 'a #/. b'
0.0157688
   6!:2 'a tally/. b'
0.154675
```

What happened is that `#/.` is handled by special code in the interpreter, but `tally/.` is not. The fact that `tally` is defined to be `#` is immaterial: the interpreter doesn't know that at the time it creates the anonymous verb for `tally/.` . The penalty is an almost-tenfold increase in execution time.

```
   6!:2 'a tally f./. b'
0.0167351
```

By flattening `tally`, we cause it to be replaced by its definition `#`, and then the special case `#/.` is recognized.

```
   b =. 0 = i. 100000
   6!:2 '(# i.@#) b'
0.0011616
   6!:2 '(tally index@tally) b'
0.00385943
```

Another example: `(# i.@#)` is handled by special code, but the names prevent the interpreter from recognizing the situation.

```
   6!:2 '(tally index@tally) f. b'
0.0011861
```

If we flatten every verb, we get good performance, but what an effort! It's much better to use the J primitives directly, so the interpreter can do its job effectively.

---

40. Explicit-To-Tacit Converter

Q: What's the most common cause of blindness among J programmers?

A: Conjunctivitis.

In the early weeks, complex tacit definitions are torturous to write and next-to-impossible to read; blurred vision and sleepless nights are the occupational hazard of the programmer who dives into tacit J. If a co-worker is banging into doors as he stumbles to refill his tankard of coffee, here's how to check him out: hold up three fingers and ask him how many he sees. If he says "three", he's merely fallen in love or is working to a deadline, and he will recover. But if he replies "I see a list of 3 items each of which is a finger", you can expect to start receiving emails that look like they've been encrypted.

You can offer as a temporary palliative J's mechanism for converting an explicit definition to a tacit one. You request the conversion by using a left operand to `:` in the range `11` to `13` instead of `1` to `4` .

```
   9!:3 (5)  NB. Do this once to select simplified display

   3 : 'x. - y.'
3 : 'x.-y.'
```

We defined a verb, and since we didn't assign it to anything, we see its value, which is just what we defined it to be.

```
   13 : 'x. - y.'
-
```

by using `13` instead of `3`, we ask the interpreter to try to find a tacit equivalent, which it did.

Here is another way to define the verb **`enclose`** from the previous chapter:

```
   13 : '(>{.x.) , y. , (>{:x.)'
([: > [: {. [) , ] , [: > [: {: [
```

The interpreter likes to use `[:` in its tacit definitions. Note that you use `13 :` to get an equivalent for both monadic and dyadic verbs; there is no `14 :` .

I recommend that you use `13 :` as your first choice for defining tacit verbs. It will find all tacit equivalents that can be generated systematically, and the explicit definition is much easier to read than a tacit definition. You will still have to use your tacit-definition skills for irregular cases, such as

```
   13 : '+:^:x. y.'
4 : '+:^:x. y.'
```

If `x.` or `y.` is used as an operand to a conjunction, as in this example, the tacit converter is not going to be able to make a tacit equivalent. The result of `13 :` is still a verb performing the requested function, but it is an explicit definition rather than a tacit one. Note that the interpreter saw that the definition contained a reference to `x.`, so it made the verb a dyad.

```
   2 (13 : '+:^:x. y.') 3
12
```

This verb applies monad `+:` to `y.`, `x.` times. Knowing about `u^:v`, you might find the tacit equivalent `+:@]^:[` :

```
   2 (+:@]^:[) 3
12
```

Special Verb-Forms Used in Tacit Definitions

It is impossible for a tacitly-defined verb to route one of its operands to an operand of a modifier inside the verb. For example, if we want a verb to set element `x` of `y` to `0`, we can try

```
   13 : '0 x.} y.'
4 : '0 x.}y.'
```

but we see that the converter was unable to find a tacit form.

Some modifiers have exotic forms that circumvent this problem. One such is the adverb } which supports the form `` x value`selector`operand} y `` . This produces the same result as `(x value y) (x selector y)} (x operand y)` so we could write

```
   a =. 13 : 'x. 0:`[`]} y.'
   a
0:`[`]}
   3 a 9 7 5 3 1
9 7 5 0 1
```

Other such forms are `x m&v y` and `x u&n y` which are equivalent to `m&v^:x y` and `u&n^:x y` respectively.

---

41. Common Mistakes

Mechanics

Watch out for Adjacent Numbers

```
   >:"0   1{y
```

fails because the 0 and the 1 are considered part of the same list. Use `>:"0   (1 {y)` or `>:"0 ] 1{y` .

Names in Sequence Do Not Form a List

`0 1` is a 2-element list, but `0 y` is an error even if `y` has the value `1` . Use `0,y` .

Remember Right-to-Left Evaluation When You Use J as a Desk Calculator

J makes a great desk calculator, but you have to remember to translate from mathematical notation to J correctly. `5 - 2 + 1` is `2`, not `4` .

How to Remember the Monads `{. {: }. }:`

Remember that `{` means *take* (`x` takes from `y`), so `}` must be *drop*. The single-dot means *beginning*, and the double-dot means *end*. So, `}.` means 'drop the first item'.

How to Remember `#.` and `#:`

The single-dot produces a single (atomic) result; the multiple-dot produces a list.

Remember the Operand Order in `e.` `i.` and `|`

The normal J convention is that a dyad's `y` operand is the one that is more 'data-like' and its `x` operand is more 'control-like'. Thus, in `x i. y`, `x` is a table and `y` is one or more values to be looked up in the table. By this convention, `x e. y` is backwards: `y` is the table and `x` is the values.

Similarly, `x | y` seems backwards from `x % y` .

Leave a Space Before `:` When Used Alone

When you use `:` or `.` as a conjunction, you must remember to leave a space before the `:` or `.` so it will not be interpreted as an inflection. The following are all errors: **`3:0`**, **`(+/ % #):*`** , and **`+/.*`** .

Use `=:` for Assignments Within Scripts

Code that runs perfectly well from the keyboard may not work if you put it into a script (.ijs) file. The problem is usually that you have used **`=.`** for some assignments. Entered from the keyboard, **`=.`** gives a public assignment, but to get the same effect in a script, you need to use **`=:`** .

Pasting Into an .ijx Window Does Not Execute

Remember that when you paste a block of text into an .ijx window, that text shows up in the window but it is not executed. To execute the text, you need to select it and then use Run|Selection.

Don't Mix `elseif.` and `else.` in the Same Structure

The control structure **`if./do./elseif./do./else./end.`** is not legal. Once you have used an **`elseif.`** you are not allowed to code **`else.`**; use **`elseif. do.`** instead of **`else.`** (the omitted condition always tests true).

Programming Errors

Remember the Asymmetry of Dyad `;`

`x ; y` always boxes **`x`**, but it boxes **`y`** only if **`y`** is unboxed. This gives you what you want when the operands are unboxed:

```
   1 ; 2 ; 3
+-+-+-+
|1|2|3|
+-+-+-+
```

But when the operands are boxed, you may be surprised at the result:

```
   (<1) ; (<2) ; (<3)
+---+---+-+
|+-+|+-+|3|
||1|||2|| |
|+-+|+-+| |
+---+---+-+
```

To have the last operand boxed, you should box it explicitly:

```
   (<1) ; (<2) ;< (<3)
```

```
+---+---+---+
|+-+|+-+|+-+|
||1|||2|||3||
|+-+|+-+|+-+|
+---+---+---+
```

Don't Give Two Operands to a Monadic Verb

When you start writing long tacit programs, you are likely to have trouble keeping track of whether a verb is being executed monadically or dyadically. Suppose that **`x`** is a set of observations and **`y`** is a set of weights, and you want to weight each observation and divide by the total weight. You might try

```
   x (* % +/) y
```

but that isn't right—the sum of the weights is **`+/ y`** and this is going to execute **`x +/ y`** . What you meant was

```
   x (* % +/@:]) y
```

Some rules to remember: a train comprising an **odd** number of verbs is a **fork**, which can be invoked monadically or dyadically. The first, third, etc.… verbs, including the last, are all executed with the same valence as the fork itself, and their operands are all the same, namely the operands of the fork itself. The second, fourth, etc.… verbs are all executed dyadically, with operands that are results of other verbs in the train.

A train comprising an **even** number of verbs is a **hook**. The first verb in the hook is always executed dyadically; the rest, taken as a train, are executed monadically on the **`y`** operand of the hook.

If any part of your train requires simultaneous access to the **`x`** and **`y`** operands of the train, you must make the train a fork rather than a hook.

Use `@:` Unless @ Is Necessary

`u@:v` has infinite rank, while **`u@v`** has the rank of **`v`** . If you don't see what difference this makes, you should drop what you are doing and read the chapter on "Compound Verbs". For practical programming, you should be in the habit of using **`@:`** unless you need **`u`** to be executed on each individual result-cell of **`v`**, in which case you may use **`@`** . The most common uses of @ are **`u@>`** to execute **`u`** on the contents of each box in a list, and **`<@v`** to box each individual result-cell of **`v`** .

Put Parentheses Around Compounds Inside Other Compounds

A modifier is greedy about what it takes for its left operand, hoovering up everything until it hits a left parenthesis or unmodified verb. For example, if you want to add 2 to `y` and then double the result,

```
   +: @: 2&+ y
```

is going to disappoint you, because it is executed as `(+:@:2)&+ y` which isn't even legal. You meant `+ @: (2&+)` . Be liberal in your use of parentheses inside compounds. You may omit the parentheses around the leftmost compound—another way to write the above would have been `2&* @: (2&+)`—but you won't regret putting parentheses around all compounds, especially when you go to change your code.

I find it easy to forget the parentheses when one of the verbs is something like `+/` that I use so much that I think of it as a primitive. When something like `>: @: +/` fails I am brought back to reality and I remember to write `>: @: (+/)` .

A Verb Is Always Executed, Even If Its Operand is Empty

J's primitives are defined to produce reasonable results when given empty operands. You should do as well with the verbs you write. Remember that if a verb has an operand with no cells, the verb is still executed once, on a cell of fills. The chapter "Empty Operands" explains what happens.

Dyad `-:` Does Not Check the Type of Empty Operands

Be aware that getting a result of `1` from `x -: y` does not guarantee that `x` and `y` are equivalent. If they have the same shape and are empty, they are considered to match even if they have different types. Subsequent operations such as `{.` would reveal the fact that the values are different, even though they 'match'.

`x u/ y` Applies `u` to Cells of `x`

To get `x u/ y` right, remember that it applies `u` to **cells** of `x` and the entire `y` . The rank of the cells of `x` is given by the left rank of `u`; use `u"n/` to set the cell-rank of `x` .

Modifiers That Refer To `y.` Have Monadic and Dyadic Valences

If your modifier contains `x.` or `y.`, its text defines a **verb** which is executed when its noun operands are known. This verb, like any explicit verb, can have both monadic and dyadic versions, separated by a line containing just a `':'` character. If you want the modifier to have a dyadic form, you must code one (by default the verb will be monadic only).

Tacit Code Does Not Simply Replace a Name By Its Definition

It is easy to develop an incorrect mental picture of how tacit programs are executed. One common error is to think that names are replaced by their definitions before a sentence is executed, in other words that if you have

```
   plus =: +
```

then a sentence

```
   1 plus 2
```

is converted to `1 + 2` and then executed. This notion immediately leads to confusion when you encounter

```
   mean =: +/ % #
```

and you expect

```
   mean 1 2 3 4 5
```

to be executed like

```
   +/ % # 1 2 3 4 5
```

which is not how it works. If you find yourself making this error, read the chapters on Tacit Programming to learn what really happens.

As a stopgap, you can imagine that each name's value is enclosed in parentheses before it is substituted. This still isn't exactly right but it gets you the right result in all situations you are likely to encounter. You would imagine that the sentence above is executed as

```
   (+/ % #) 1 2 3 4 5
```

which gives the correct answer. You must realize that `(+/ % #)` is a *fork*, with its own rules for processing its operands.

<<   >>   Contents   Help

42. Valedictory

You have learned enough J to understand J programs and to put your own ideas into J. I hope you will now do two things: practice using J so that J becomes the language of your mind's ear, the way you naturally express algorithms; and read the Dictionary with care, so you can learn J fully. It'll be like weight training: in a few months you'll look in the mirror and be amazed at the programmer you've turned into.

Good luck, and I hope to see you on the J Forum!

<< >> Contents Help

43. Glossary

Adverb One of the primary parts of speech. An adverb modifies the word or phrase to its left to produce a derived entity that can be any of the four primary parts of speech.

Anonymous Having no name. Said of the result of an execution.

Atom Any single number, single character, or single box. Also called a scalar. An atom is a noun with rank `0` .

Array A noun comprising atoms arranged along one or more axes. Each atom is identified by its index list. J arrays are rectangular, meaning that all 1-cells contain an identical number of 0-cells, and all 2-cells contain an identical number of 1-cells, and so on.

Axis One of the dimensions along which the atoms of an array are arranged. The atoms of every noun are arranged along zero or more axes where each axis has a corresponding length and each unique list of nonnegative integers in which each integer is less than the length of the corresponding axis designates a unique atom of the noun.

Boolean Having a numeric value restricted to the values `0` and `1` .

Cell A subarray of a noun consisting of all the atoms of the noun whose index lists have a given prefix. For positive *k*, each **k-*cell*** of a noun whose shape is *s* has the shape `((- k <. #s) {. s)` and together they can be assembled to reconstruct the noun. For negative *k*, the *k*-cells are defined to be the `(0 >. k + #s)`-cells.

Conjunction One of the primary parts of speech. A conjunction modifies the words or phrases to its left and right to produce a derived entity that can be any of the four primary parts of speech.

Copula One of the parts of speech, signifying an assignment of a value to a name. The copulas are `=.` and `=:` .

Derived Entity The result of executing an adverb or conjunction. If the part of speech of a derived entity is known, it may be called, for example, a derived verb.

Dyad (dyadic) A verb with left and right operands (which must be nouns). Any verb may be used as a monad or dyad, depending on whether it has a left noun operand when it is executed.

Entity A noun, verb, conjunction, or adverb

Execution The process of replacing a verb and its operands on the execution stack with the result from applying the verb to those operands; or of converting a fragment into a derived entity in accordance with the definition of the fragment, and replacing the fragment on the execution stack by a single word referring to the derived entity.

Execution Stack The set of words of a sentence that have been examined by parsing in its search for an executable fragment, as modified by the replacement performed by execution.

Fill An atom appended to a noun to extend the length of the noun to a required length, especially when the noun must be extended because it is being made a cell in an array whose cells are longer than the noun.

Fragment 2 or 3 words on the execution stack in a context that makes them executable.

Frame The ***frame of a noun with respect to* k*-cells*** is the shape of the noun with the last *r* items removed, where *r* is the rank of a *k*-cell of the noun. When a noun is viewed as an array of *k*-cells, the frame with respect to *k*-cells is the shape of the array of *k*-cells.

Framing Fill A fill added when the results from applying a verb on its operand cells are being joined into an array. Framing fills are always `0`, `' '`, or `a:` depending on the type of the result.

Fret A marker indicating the start or end of an interval.

Functional Programming A method of writing programs that describes only the operations to be performed on the inputs to the programs, without the use of temporary variables to store intermediate results. J's tacit programs are an example of functional programming. Aficionados of functional programming consider it to be a purer statement of an algorithm than the usual statement in a procedural language; as the expert J programmer Randy MacDonald has said, "If you're not programming functionally, you're programming dysfunctionally".

Global Of a name, accessible as a simple name by any verb. In the J Dictionary the word 'global' has the meaning we have assigned to the word '**public**'.

Index In an array, an integer indicating position along an axis. The index of the first atom along an axis is `0` .

Index List A list of integers with the same length as the shape of a noun, designating an atom of the noun by giving its position along each axis.

Interval A sequence of consecutive indexes or cells.

Item A _1-cell of a noun. An array is a vector of its items. An atom has one item, itself.

List An array of rank `1` .

List of An array whose items are; as in 'list of 3-item lists'.

Local See **private**.

Locale A namespace in J. The locale in which a public name is defined is an attribute of the name. A locale is identified by a locale name which is a character string not containing an underscore.

Locative A name including both a simple name and an explicit locale.

Modifier An adverb or conjunction, which modifies its operand(s) to produce a derived entity.

Monad (monadic) A verb with no left operand. Any verb may be used as a monad or dyad, depending on whether it has a left noun operand when it is executed.

Name Either a simple name or a locative, to which an entity can be assigned.

Noun One of the primary parts of speech. An atom or array of atoms. Nouns hold the data that verbs operate on.

Parsing The right-to-left search for suitable patterns in a sentence. When a suitable pattern is found, that subset of it that constitutes an executable fragment is executed.

Part of Speech One of the six categories into which words are classified; or, the word or entity so classified. Every word has a part of speech: for primitives, the part of speech is defined by the language; for names, the part of speech is that of the entity assigned to the name. The parts of speech are: noun, verb, conjunction, adverb, punctuation, and copula. The primary parts of speech are noun, verb, conjunction, and adverb.

Partition A selection of (possibly noncontiguous) items of an array, brought together as the items of a new array to be operated on by a verb.

Path See **search path**.

Primitive A word whose meaning is assigned by the J language.

Private Of a name, assigned in the namespace of an explicit definition and accessible only within the explicit definition in which it was assigned.

Public Of a name, assigned in a locale and accessible from any locale via a locative.

Punctuation A part of speech. Punctuation is not executed but it affects the execution of other words. Punctuation in J comprises **`(`**, **`)`**, **`NB.`**, and control words.

Rank Of a noun, the number of axes along which the atoms of the noun are arranged; the number of items in the shape. Of a verb, the highest rank of the noun operands that the verb can operate on.

Scalar See **atom**.

Script A file containing J sentences.

Search Path Of a locale ***l***, the list of the names of locales that will be searched to find the definition of a name originally sought but not found in *l*.

Sentence An entire executable line.

Shape The list of the lengths of the axes of a noun.

Simple Name A list of letters, numbers, and underscores beginning with a letter, used to refer to an entity.

Train A sequence of words that cannot immediately be executed to produce a noun result.

Type An attribute of a noun: numeric, literal (also called string), or boxed.

Valence Of a verb definition, an indication of the number of noun operands that the definition can accept: monadic if 1, dyadic if 2, dual-valence if either 1 or 2.

Vector A sequence of cells arranged with a leading axis. An array can be construed as a vector of its items.

Verb One of the primary parts of speech. A verb operates on the noun to its right (and its left, if a noun is to its left) to produce a noun result.

Verb Fill Fill added during processing of a verb. The fit conjunction **`!.f`** can often be used to specify the fill atom to be used for verb fill.

Word A sequence of characters in a sentence, recognized as a lexical unit. A word is either a name, a primitive, a constant (which may be a number or a character or a list of either), or a synthetic word used to refer to the result of an execution.

<< >> Contents Help

44. Error Messages

When J encounters an error executing a sentence it stops and displays the sentence. The interpreter removes any excess spaces from the sentence and then adds three spaces before the word whose execution triggered the error. For example:

```
   2 3 + 0 1 2 * 3 4 5
|length error
|   2 3    +0 1 2*3 4 5
```

The error occurred during the execution of the `+` verb.

The errors you are most likely to encounter are:

control error You have an incomplete control structure, for example an `if.` without matching `do.`/`else.`/`elseif.` and `end.` .

domain error An operand has a value that is not allowed, for example a string operand to an arithmetic operation, or an out-of-range numeric left operand to dyad `o.` . One common source of domain error is trying to execute a verb when no definition exists for the valence (monadic or dyadic) that you are trying to execute.

Errors encountered during execution of `wd` are reported as domain errors.

file name error You specified a file name that is invalid, or attempted to read a nonexistent file.

file number error You specified a number that is not the number of an open file.

ill-formed name You used an illegal name, such as `name_1ff_` (illegal because `1ff` is not a valid locale name)

ill-formed number You used an illegal number such as `14h` . A word that starts with a numeric character must be a valid number, and vice versa.

index error You attempted to access an element outside the bounds of an array.

length error You used a dyadic verb with operands that did not agree (i. e. one frame was not a prefix of the other). Or, a verb expected an operand of a certain length and you gave an incorrect length (for example `1 2 3 {. 5 5`)

limit error You exceeded one of J's limits, for example by specifying a comparison tolerance greater than `2^_34` . The most common cause of a limit error is an infinite recursion that exhausts the available stack space.

nonce error You tried to do something reasonable, but the system doesn't support it yet. So, for the nonce, find another way to do it.

open quote Your sentence contains an unmatched single-quote.

out of memory The interpreter asked the operating system for enough memory to fulfill your request, but the operating system refused. You need to use smaller nouns.

rank error You specified an operand with an invalid rank.

spelling error You typed an erroneous `.` or `:` to produce a meaningless word like `+..` or `fred.` .

syntax error Your sentence contains an invalid sequence of parts of speech, as in `5 +` . Or, you have an explicitly-defined verb whose last-executed sentence gives a result that is not a noun: that would make the verb have a non-noun result, which is intolerable.

value error You have asked the interpreter to evaluate a name that has not been defined. There is more to this definition than meets the eye. A noun, adverb, or conjunction is evaluated when it is encountered during the right-to-left execution of a sentence. A verb is evaluated when (a) it is executed with its noun operand (s) or (b) when the name of the verb is typed as the only word in a sentence, at which time the verb is evaluated for display purposes. For example, the sentence

```
   undefname
```

will result in a value error, because you are asking the interpreter to display the value of the undefined name. However, the sentence

```
   name =: undefname
```

will not fail, because `name` is defined to be a reference to `undefname`, and `undefname` does not have to be evaluated (the undefined name is assumed to refer to a verb of infinite rank that will be defined later). Subsequently,

```
   name
undefname
```

`name` is defined, but if we force the interpreter to use it:

```
   name 5
|value error: undefname
```

```
|       name 5
```

the underlying undefined name is exposed.

An important case is:

```
   undefname1 undefname2
undefname1 undefname2
```

Note that this did not result in a value error. Recall that undefined names are assumed to be verbs; we defined a hook from the two presumed verbs and then asked the interpreter to display the hook. The interpreter was able to do that without evaluating either name. Either name by itself would produce a value error.

If you write an undefined name as the only word of a sentence in a script, the interpreter will have no need to evaluate the name (since it doesn’t have to display the result), and the sentence will be ‘executed’ without error.

<<   >>   Contents   Help