# Errors in J

Justus Perlwitz

2020-07-31

## Contents



For the past month I've been using the J programming language to solve Project Euler puzzles. The language makes many of the tasks that one would usually spend quite some time on, especially with multidimensional arrays, quite pleasant to work on. It has given me enough of a motivational boost to finally solve 50 puzzles in total, and with J in my toolkit I don't see myself stopping there.

## 1 Embracing Weirdness

On the other hand, working with J also makes me realize what an obscure and *different* tool I've stumbled upon there. J is associated with Kenneth E. Iverson, one of the most influential pioneers in programming language design who has previously created APL. Kenneth E. Iverson and Roger Hui created the language J in the 90s.

Considering the programming language landscape at that time, J and its predecessor APL must have been miles ahead of so many other languages out there. Only in the last ten years with the advent of array programming paradigms in other languages, for example Numpy in Python, one might think that the advantages of thinking in arrays are permeating the rest of the world.

J is a language that handles arrays of any dimension as first-class objects and provides an extreme wealth of primitive operators that can be combined in almost any way thinkable. A good example of the power of operators is the mighty *under* verb. It allows you to apply two functions v and u in sequence while automatically deducing the inverse of v (!). "u under v" will be evaluated as $v^{-1} \circ u \circ v$.

A concrete use case: Turn an integer into a string, sort the digits, and turn it back into an integer. An easy task for *under*.

Doing this as a one-liner in Python 3 would roughly look like this

```
>>> [int(a) for a in ("".join(sorted(str(a))) for a in [321, 432])]
[123, 234]
```

If we did this in J, turning a number into its digits can be done by inverting the base operator. While that sounds like proposing to use the reader monad to add two integers in Haskell, it's really not as bad as it sounds. This is our v function.

```
   10 #.^:_1 ] 321 432
3 2 1
4 3 2
```

Sorting each digit can be done by using the grade up or grade down operator. Since the result is not a sorted array, but just the positions of the elements *if it was sorted*, we have to apply it to the array itself.

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

And using a fork we can apply the result back on the original list of digits. This is our u function.

```
   (/: { ]) 3 2 1
1 2 3
```

Combining u and v using the under operator `&.:`, and adjusting for rank of the right-hand operand, we arrive at:

```
   NB. u   under  v
   (/: { ])&.:(10&#.^:_1)"0 ] 321 432
123 234
```

I don't want to give the impression that getting to this point was particularly easy for me. A lot of Project Euler puzzles for example involve decomposing numbers into their digits. And if you've done it a few times, it's easy to write. But the first few times I had to deal with errors that I just couldn't understand, especially because the documentation rather light on this subject.

And perhaps with this article I can make it a little bit easier for you as well to troubleshoot common errors in writing J programs.

## 2 Dealing with Errors

Just like every other part of J feels *terse* and *tacit*, so do errors have a little bit of mystery and magic about them. My experience with errors is usually:

1. Somehow I stumble upon a length, domain, or rank error.
2. If I'm working with tacit expressions, I repeatedly add `[:` caps or check parentheses until I'm sure everything is right.
3. I check verb ranks and add copious amounts of rank adverbs until the result looks right.
4. I go over my usage of conjunctions in the `@` family one more time and make sure I really understand the flow of data correctly.

And then somehow most of the time it resolves the problem. But then something kept bugging me: I encounter all those errors, but I don't actually know how to *deliberately* cause them. If I know how to deliberately bring about a dreaded domain error, then surely I can more confidently solve my issues and the four-step process shown above might become a little easier.

We will take a look at all the errors I have encountered so far, and then show a minimal piece of code that will deliberately cause them. Then we will discuss mitigation strategies for each error and in the end have the cathartic experience of having learned something new about this wonderful language.

As a side note, my J version is

```
9!:14 ''
j901/j64avx2/darwin/release-f/commercial/www.jsoftware.com/2020-06-11T16:07:02
```

# 3 Cataloging Errors

Errors I typically encounter with J are (ranked by how vexing they are):

1. domain errors,
2. length errors, and
3. rank errors.

A comprehensive overview of all error message can be found here.

First, I want to explain how J error messages are formatted. Knowing their formatting makes it slightly easier to dissect where exactly the error was caused.

# 4 About Error Messages

Let's use type incompatibility as an example. This will trigger a domain error, as we will see below. Say, we have this code and run it in using `jconsole`.

```
1 , 'hello'
```

This code snippet will append the integer 1 to the string 'hello'. Since J doesn't know what to do with that, it will give us a domain error like so

```
   1,'hello'
|domain error
|   1    ,'hello'
```

Do you notice that J added four spaces between the integer and the append verb `,`? The J interpreter uses this to hint at the location where the error occurred. This tells us that the problem happened when applying the integer 1 to the append operator, as the white space can be found exactly between the two words.

# 5 Domain Error

We've already seen that a domain error can be caused by trying to add incompatible types into the same array. The *type* foreign verb can be helpful when finding out what types you are dealing with and is helpful as a diagnostic tool. For example, we can run

```
   3!:0 ] 1
1
```

Referring to this table we can then find out that result 1 means the data we passed it is boolean. That means J interprets `1` as boolean, not as an integer, here. For a string, we will get the following value:

```
   3!:0 ] 'hello'
2
```

Referring to the same table, we find out that 2 indicates that this list contains bytes.

But this is not the only place where we can hit domain errors. The cap verb `[:` helps when creating tacit expressions. Within a fork it indicates that this part of the fork should never be evaluated. This is a specific verb that the J interpreter knows how to evaluate correctly, but only if it is a part of a fork. Calling it any other way will simply result in a domain error.

If you are writing a tacit expression, this can happen by accident. Given three verbs u, v and g, let's say you're editing an expression that contains several caps like so

```
[: u [: v g
```

This expression contains two forks, with both of them being capped on the left side. Now you do some shuffling around and editing, and you end up with

```
[: [: v g
```

Running this on a noun would result in a domain error.

To give you a concrete example, this code will run fine:

```
   ([:-[: + *) 1
_1
```

Here, u, v, and g are `-`, `+` and `*`. And perhaps for a second we don't pay any attention and remove the left `-` operator without removing the cap on its left side. Running this will then lead to the dreaded domain error.

```
   ([: [: + *) 1
|domain error
|       ([:[:+*)1
```

There we have our domain error again. Unfortunately this can happen quite easily if you have a deeply nested expression. I'm not too confident at parsing long J expressions yet, and once a line exceeds roughly 30 characters I have a hard time checking syntactic/semantic correctness without actually running the code. I am confident though that this is just a matter of exercise.

In order to reduce this kind of error, it is useful to

1. think about how the J interpreter reads through your code and how it parses and evaluates forks, and
2. have a good mental model of the data that is flowing through your expressions and at what point they have what type

Since so many things in J are implicitly assumed by the interpreter, such as forks and type casts, and explicit casts or variable definitions are hardly possible, keeping track of the data flow in your mental model becomes even more important. Of course the great advantage is that you can have incredibly short code that does an incredible amount of stuff.

On the other hand, many developers consider

```
int c = 1;
```

to be more than just a verbose variable and type declaration, but documentation that serves to make this piece of code readable in the years to come. Ultimately it comes down to what kind of project you're using a language for and what the demands are.

If I hit a domain error while solving a Project Euler puzzle, it's not a big deal. Domain errors on a deployed web application? Troublesome.

# 6 Length Error

In J it is important to be aware of the concept of Agreement. It is used in the context of dyadic verbs. Given an expression `x v y`, one would say that x and y are in agreement when they match according to the rules of agreement.

A few simple examples that show how J sees agreement:

```
   NB. same rank
   1 2 3 + 4 5 6
5 7 9
```

Here we add the atom 1 to the list 1 2 3:

```
   NB. 1 is broadcast
   1 + 1 2 3
2 3 4
```

And even this works. Note the shape of each noun:

```
   1 2 + i. 2 3
1 2 3
5 6 7
   NB. the rank of each noun is
   $ 1 2
2
   $ i. 2 3
2 3
```

And we immediately run into trouble when trying to do the opposite:

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

Why is that? Can't we just add 1, 2, and 3 to each row of `i. 2 3`?

Simply put, J expects that the shapes of both operands have matching prefixes. Since `1 2` has the shape `2`, and `i. 2 3` has the shape `2 3`, the two can be matched when adding them together with `+`. The `+` verb has rank `0 0` in dyadic use, and therefore works on individual atoms of x and y. The documentation furthermore guarantees that the result will have the *same shape*.

On the other hand, when we use `1 2 3` as our left-hand operand, its shape `3` does not have a common prefix with the shape of `i. 2 3`, namely `2 3`. Therefore, J does not know how to correctly perform the operation.

In J, this is called cell matching. For example, in

```
   1 2 + i. 2 3
```

J matches `1` with `1 2 3`, and `2` with `4 5 6`. Finally, after performing the addition on the matched cells, it reassembles the result and returns it. There are a lot more complications like framing fill, which is why the article on agreement on the J wiki is incredibly helpful.

When I encounter length errors, I first ensure that I am not trying to combine data that is obviously not meant to be combined. Like a list of rows with a list of columns.

Then, after making sure that the correct data is being operated on, it's a good idea to see whether cells are being matched correctly. While `1 2 3 + i. 2 3` can not be matched, we can modify the verb to make it agree. For this, we have to use the rank conjunction. When applied correctly, we receive the expected result:

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

Consider also that the rank conjunction can be applied for monadic and dyadic invocation separately.

As an example, if you want to find the remainder for each number from 0 to 9 when divided by 2, 4, or 8, you could try to run the following, and will run into a slight problem.

```
   2 4 8 | i. 10
|length error
|   2 4 8    |i.10
```

We can find out what is happening by applying the verb info `b.` and seeing what noun ranks it accepts:

```
   NB. Return the monadic rank first, and after that the monadic ranks
   | b. 0
0 0 0
```

When invoking `x | y`, the verb works on atomic cells of `x` and `y`, as indicated by the `0 0`. What we want instead, is for an atom of `x` to operate on a list of `y`, so for `2` to be matched with `0 1 2 3 ...`. Therefore, we can modify `|` with the rank modifier again and see it working:

```
   2 4 8 |"0 1 [i. 10
0 1 0 1 0 1 0 1 0 1
0 1 2 3 0 1 2 3 0 1
0 1 2 3 4 5 6 7 0 1
```

Exactly the result we wanted. We learn that understanding rank is incredibly important. The challenge of learning and understanding rank leads us to the next section.

# 7 The Importance of Understand Rank

Rank is one of those concepts of J that feel quite easy, once you've understood it, but is incredibly intimidating on a newcomer. Rank can have two meanings:

1. The rank of a noun, as in the number of dimensions that a noun has. A single number has the rank 0, a list has the rank 1, a table rank 2, and so on. This is more or less the same as the *dimensionality* of an array in other programming languages.
2. The rank of a verb indicates the highest rank of its operand nouns. For example, a verb of rank 2 indicates that the highest rank of it's operand can be 2 and therefore a table. A verb can also have rank *infinity*, indicating that it can operate on nouns of any rank. And since J has the concept of monadic and dyadic verbs, a verb can have a rank for its monadic invocation, and a rank consisting of two numbers for its dyadic case. In that instance, the first number indicates the highest rank of the left operand, and the second number indicates the highest rank of the right operand. Dyadic `+` as an example has the rank `0 0`, meaning that `x + y` operates on individual atoms of `x` and `y`.

I wonder if rank could have been easier to understand if it was called "dimensionality", since this is something most developers should be familiar with. On the other hand, rank alludes to J's mathematical background, and someone perhaps familiar with linear algebra rank might feel more at home with calling it rank.

As we've covered in the previous section, J uses some really clever rank matching to establish agreement.

When your noun ranks do not match you will most likely encounter a *length error*, since J wants to match cells, and is unable to because the cells on the left and on the right do not match up *because of their shape and rank*.

# 8 Rank Error

Even though the name implies it, the rank error does not mean that two dyadic operands have mismatching rank. The rank error is much more trivial. For example, any verb that expects the left argument to have a certain rank, and instead receives something else, will result in a rank error being thrown.

The primes verb `p:` is great for solving Project Euler puzzles that involve prime numbers. Monadically, it will tell you the nth prime number, and dyadically it can do a lot of things, such as give you the number of prime numbers that are smaller than y. Used dyadically, x tells `p:` in which mode to operate. We can test a number of integers for primality by running

```
   1 p: i. 9
0 0 1 1 0 1 0 1 0
```

If we want to do two things at the same time, such as testing for numbers that *are not* prime, and numbers that *are* prime, then we run into trouble again:

```
   0 1 p: i. 9
|rank error
|   0 1     p:i.9
```

Looking at the documentation for `x p: y` here, we find that for its dyadic use, it expects

> Rank Infinity – operates on x and y as a whole

And looking up the meaning of rank infinity further, reveals that

> This verb operates on x and y in their entirety, producing a single result which may have any rank or shape, depending on the individual verb.

This tells us that how x and y is up to the verb's discretion. The description of *rank infinity* linked above shows a list of many other verbs that all have need you to take a close look at each verb's documentation if you are unsure about the required shape.

What are we to do if we *really* want to produce the desired result from above: Return two lists, one containing 1's for each number that isn't prime, and one containing 1's for each number that *is* prime. How can we avoid the rank error from before?

We can simply apply the rank modifier like in the previous domain error section, and try our luck. Our intention is to:

1. Do not change how the right side is operated on
2. Change `p:`'s behavior to treat the left side x as a list and operate on each cell – atom in this case – separately.

Not changing the right side would mean we would have to keep the rank at infinity, or as written in J as `_`. Since the left-hand side is a list and we want to operate on its atoms, we want to work on it with rank 0. We modify the previous attempt as follows and receive the correct result:

```
   0 1 p:"0 _ i. 9
1 1 0 0 1 0 1 0 1
0 0 1 1 0 1 0 1 0
```

J, as a *right-to-left* language, is easy to write when data flows from right to left. It becomes more challenging in dyadic use cases where you want to control how data from the left *and* right is matched.

When dealing with rank errors it helps to thoroughly read the documentation of the verbs you are dealing with. In the case of dyadic verbs I try to be extra careful to understand the rank requirements for both operands and whether their shapes agree or not.

With everything, if you apply enough practice it will become second nature.

# 9 Conclusion

These are some of the problems I most frequently encounter when working with J, especially when dealing with numerical problems. If you have any errors that you frequently encounter, or want to share effective troubleshooting strategies that you like to use when working with J, I would sincerely appreciate your feedback.