JWithATwist Reference Manual

Table of Contents

JWithATwist

JWithATwist is a programming language under development. It has many similarities with J. You can find descriptions of J at www.jsoftware.com.

I use a J-like syntax to describe this language. The terms can have slightly different definitions because of language differences.

The most important language concepts are:

- *Noun* - an n-dimensional array of values where the values are simple data types or other n-dimensional arrays.
- *Verb* - a function with one or two noun arguments returning a noun.
- *Adverb* - a function with a verb argument and one or two noun arguments, returning a noun.
- *Conjunction* - an operator with two verb arguments and one or two noun arguments, returning a noun.

A *monadic* verb has one argument, a *dyadic* verb has two. Adverbs and conjunctions are also called monadic or dyadic after their number of noun arguments.

Nouns, verbs, adverbs and conjunctions all have their own sections in this manual. A section describes the language syntax. The last section describes the use cases.

The JWithATwist Noun

The noun is n-dimensional array. It has one of these data types:

- Integer
- Float
- Boolean
- Unicode
- Boxed

Integer, Float, Boolean and Unicode corresponds to the .NET data types of Int64, Double, Bool and Char, Unicode-16.

A noun of data type *boxed* contains other n-dimensional arrays, which can also be boxed.

The *rank* of a noun is the number of dimensions of the array it contains.

The nouns are immutable. The value is defined when it is created. It can not be changed. Modification operations return a new noun.

A noun with zero dimensions, a singleton, is called a *scalar*.

The noun is seen as a structure of something called *k-cells*. The scalar is a 0-cell, the vector a 1-cell, the table a 2-cell and so on for higher dimensions. An n-cell contains a number of (n-1)-cells which contains a number of (n-2)-cells.

A 0-cell can also be called a *cell*.

Literals

Simple data types are entered in the following ways:

- Integer: `0 -1 2 3`
- Float: `0.0 -1.0e4 nan infinity -infinity`
- Boolean: `true false`
- Unicode: `"John" "George"`

If you enter only one value the result is a scalar, a 0-cell. If you enter more than one value the result is a vector.

When you enter unicode values the result is a boxed noun of unicode nouns. A boxed array of unicode arrays.

The JWithATwist Verbs

The JWithATwist verbs are normally array operations. The arguments are one or two arrays, the result another array. You write `1 5 + 2 3` and get the result `3 8`.

We talk about the *rank* of a verb. The rank is the k-cell or k-cells of the argument or arguments that the actual underlying operation is performed on. A scalar operation is done on one or two 0-cell arguments. We say that a monadic scalar verb has rank 0 because the underlying operation is performed on one 0-cell argument. We say that a dyadic scalar verb has rank 0 0 because the underlying operation is done between two 0-cell arguments. The arguments of the verb are arrays, but the underlying operation has a 0-cell argument or two 0-cell arguments.

To show that a verb operates on the whole of an argument, we denote the rank with _ . A monadic verb operating on the whole of its argument has rank _ . A dyadic verb operating on the whole of its both arguments has rank _ _. JWithATwist does not allow you to write this in the program, but you can use the actual rank of the argument or any higher number.

Most verbs are scalar, which means that the underlying operation is done between corresponding cells of the argument arrays. The result of the underlying operation is another cell. The array part of the operation is handled in the same way for all of these verbs. The verbs use a helper program to handle the loop over the parts of the arguments the verb does not handle.

Other verbs have other ranks. The underlying operation can be performed on part or whole of the array arguments. If the underlying operation is not done on the whole of the arguments, the verbs use a helper program to handle the loop over the parts of the arguments the verb itself does not handle.

There are four such helper programs, one for monadic and one for dyadic scalar operations, one for monadic and one for dyadic array operations.

To understand JWithATwist it is important to understand these helper programs.

The Monadic Scalar Operation Helper Program

The monadic scalar operation helper program performs the underlying operation on each cell of the argument. The result is an array of the same size. This operation is usually called *map* in functional languages.

The monadic scalar operation helper program also handles type conversions from integer to float. If a float operation is performed on an integer it is automatically converted to a float before the operation.

The Dyadic Scalar Operation Helper Program

The basic function of the dyadic scalar operation helper program is to perform the underlying operation between the corresponding cells of the argument arrays. The result is an array of the same size. This operation is usually called *map2* in functional languages.

If one of the arguments is a scalar, the operation is performed between this scalar and each of the cells of the other argument.

If the arguments have different shape, but the shape of one of the arguments is the same as the shape of the first part of the other argument, a loop is created over the remaining part of the larger argument. We talk about *agreement*. Agreement is how JWithATwist handles operations between arguments of different shape and rank. Which combinations are allowed and which combinations are not.

Example:

```
    10 20 + |i. 2 3
10 11 12
20 21 22
```

The shape of the left argument in the example is 2, the same as the first part of the shape 2 3 of the right argument. The helper program creates a loop over the remaining part, the cells in the rows, and the corresponding part of the left argument is added to each of these cells.

The dyadic scalar operation helper program also handles type conversions from integer to float. If a float operation is performed on an integer it is automatically converted to a float before the operation.

The Monadic Array Operation Helper Program

The monadic array operation helper program is available to the user as the monadic rank adverb.

If a monadic verb is not scalar, if the underlying operation does not handle the whole of it's array argument and if the argument is correct, it calls the monadic array operation helper program to loop over the parts of the argument it does not handle.

As an example, the monadic Iota verb is an index generator. It generates an n-dimensional array of indexes when given a vector argument. `|i. 3` gives a result of `0 1 2`. `|i. 2 3` gives a shape 2 3 array with rows `0 1 2` and `3 4 5` as result. If given an argument of rank higher than 1, it calls the monadic array operation helper program to handle the loop over the rank 1 parts it handles.

The Iota verb is then called a number of times by the monadic array operation helper program, which the Iota verb itself called as a helper program. Each time the Iota verb is called with a vector argument. The result is a number of n-dimensional arrays with different shape but with the same rank, since the 1-cells of the argument all have the same length. The monadic array operation helper program calculates the maximum shape of the result in each dimension. It pads all the results so that they all have this same maximum shape. The shape of the result is the shape of the part of the argument shape the Iota verb does not handle catenated with the maximum shape of the Iota verb results.

Example:

```
    2 2 $ 2 1 1 3
2 1
1 3
    |i. 2 2 $ 2 1 1 3
Type: Integer, Shape: 2 2 3 , Values: 0 0 0 1 0 0 0 1 2 0 0 0 ...
    |$ |i. 2 2 $ 2 1 1 3
2 2 3
    |i. 2 1
0
1
    |i. 1 3
0 1 2
```

The Iota verb handles each row of the argument and calls the monadic array operation helper program to loop over the two rows. Like this: `|i. '/ 1 / 2 2 $ 2 1 1 3`.

The results from each call to the Iota verb from the monadic array operation helper program is the column `0 1` and the row `0 1 2`.

These results are padded to the shape 2 3. The final result has the shape of `2 2 3`, the part of the argument shape not handled by the Iota verb, `2`, catenated with `2 3`, the maximum shape of the Iota verb results.

The Dyadic Array Operation Helper Program

The dyadic array operation helper program is available to the user as the dyadic rank adverb.

If a dyadic verb is not scalar, if the underlying operation does not handle the whole of both of it's array arguments and if the arguments are correct, it calls the dyadic array operation helper program to loop over the parts of the argument it does not handle.

As an example, the Shape verb shapes the whole of the right argument to the shape described by it's vector left argument. `2 2 2 $ 0 1 2 3 4 5 6 7` creates a three-dimensional array with the shape `2 2 2`. If given a left argument of rank higher than 1, the Shape verb calls the the dyadic array operation helper program to handle the loop over the rank 1 parts of the left argument it handles.

The Shape verb is then called a number of times by the dyadic array operation helper program, which the Shape verb itself called as a helper program. Each time the Shape verb is called with a vector left argument. The result is a number of n-dimensional arrays with different shape but with the same rank, since the 1-cells of the left argument all have the same length. The dyadic array operation helper program calculates the maximum shape of the result in each dimension. It pads all the results so that they all have this same maximum shape. The shape of the result is the shape of the part of the left argument shape the Shape verb does not handle catenated with the maximum shape of the Shape verb results.

Example:

```
    2 2 $ 2 1 1 3
2 1
1 3
    ( 2 2 $ 2 1 1 3 ) $ |i. 3
Type: Integer, Shape: 2 2 3 , Values: 0 0 0 1 0 0 0 1 2 0 0 0 ...
    |$ ( 2 2 $ 2 1 1 3 )
2 2 3
    ( 2 1 ) $ |i. 3
0
1
    ( 1 3 ) $ |i. 3
0 1 2
```

The Shape verb handles each row of the left argument and calls the dyadic array operation helper program to loop over the two rows. Like this: `( 2 2 $ 2 1 1 3 ) $ '/ 1 2 / |i. 3 .`

The results from each call to the Shape verb from the dyadic array operation helper program is the column `0 1` and the row `0 1 2`.

These results are padded to the shape 2 3. The final result has the shape of `2 2 3`, the part of the argument shape not handled by the Shape verb, `2`, catenated with `2 3`, the maximum shape of the Shape verb results.

Monadic Scalar Verbs

The monadic scalar verbs are described in table XX, YY, ..

Table 1. Arithmetic Monadic Scalar Verbs

`|*`	Signum
`|%`	Reciprocal
`|<.`	Floor
`|>.`	Ceiling
`|-`	Negate

| |	Magnitude

Floor and Ceiling are float operations. In JWithATwist the result in an integer. The largest integer a float can correctly represent is 2 raised to the power of 51. The smallest integer a float can correctly represent is the negation of this. In JWithATwist, if Floor and Ceiling is used on a float with larger absolute value the result is domain error.

Table 2. Boolean Monadic Scalar Verbs

| - .	Not

Table 3. Conversion Monadic Scalar Verbs

| _	Boolean to integer

Dyadic Scalar Verbs

The dyadic scalar verbs are described in table XX, YY, ..

Table 4. Arithmetic Dyadic Scalar Verbs

+	Add
-	Subtract
*	Times
%	Divide
< .	Min
> .	Max
^	Power

In JWithATwist Power always returns a float.

Table 5. Comparison Dyadic Scalar Verbs

<	Less than
<:	Less than or equal
>	Greater than
>:	Greater than or equal
=	Equal
~:	Not equal

JWithATwist does not use comparison tolerance. The use of equality operations on floats is not recommended.

Table 6. Boolean Dyadic Scalar Verbs

* .	And
+ .	Or

Non-Scalar Monadic Verbs

The non-scalar monadic verbs are shown in table XXX

Table 7. Non-Scalar Monadic Verbs

`|<`	Box
`|<`	DefaultFormat
`/:`	GradeUp
`|i.`	Monadic Iota
`|>`	Open
`|$`	ShapeOf
`|#`	Tally

The Box Monadic Verb

The Box monadic verb is a verb of rank _ . It is applied to the whole of the right argument. There is no underlying verb. The Box verb creates a boxed scalar noun containing its right argument. The box verb can be used together with the monadic Rank adverb to box k-cells of the argument. It can be used in verb definition arguments to adverbs and conjunctions to box the results. If the verb definition argument to an adverb creates results of different shapes, you might have to box the results. You normally have to open boxed nouns with Open before you can operate on them. After doing your operation you can box the results.

Examples:

```
   |< 1 2 3
1 2 3
 |$ |< 1 2 3

   |< '/ 0 / 1 2 3
1 2 3
   |$ |< '/ 0 / 1 2 3
3
   "" {! |< ( |> [ ) , |> ] } / "John " "Lennon"
John Lennon
   |$ |> "" {! |< ( |> [ ) , |> ] } / "John " "Lennon"
11
   "John " "Lennon"
John Lennon
 |$ "John " "Lennon"
2
```

The DeaultFormat Monadic Verb

The DefaultFormat monadic verb is a verb of rank _ . It is applied to the whole of the right argument. There is no underlying verb. The DefaultFormat verb is also used to format the output in the JWithATwist terminal.

DefaultFormat formats data of rank less than 2 in a simple way. If given data of higher rank the output shows data type, shape and up to 20 values of the output.

Numeric and boolean data is first converted to boxed arrays. Each cell is formatted to a Unicode array and is then boxed.

DefaultFormat recursively formats data in boxed arrays. All data is top right adjusted in the area corresponding to the box.

The size of the boxes in the output is determined by the maximum number of rows of content in the boxes on the same row of boxes and the maximum number of cell columns of content in the boxes in the same column of boxes.

Examples:

```
   |': 1 2 3
 1 2 3
   |': -1.2345678e-300 1
 -1.23457e-300 1
   |': true false
 true false
   |': "JWithATwist is cool!"
JWithATwist is cool!
   |': ( "Column 1" " Column 2" ) , |< |'/ 0 / |i. 2 2
Column 1 Column 2
       0        1
       2        3
   |i. 2 2 2
Type: Integer, Shape: 2 2 2 , Values: 0 1 2 3 4 5 6 7 ...
```

The GradeUp Monadic Verb

The GradeUp monadic verb is a verb of rank _ . It is applied to the whole of the right argument. There is no underlying verb.

The result of GradeUp is an index which can be used together with From to sort the right argument in ascending order.

If you apply GradeUp to the produced index you get a new index which can be used to sort the right argument back to its original order.

This sort back functionality is frequently used in languages like JWithATwist. You sort information, process it in sorted order and sort the results back to the original order.

GradeUp can be used on 1 and 2-dimensional arrays. It can not be used on arrays of higher rank.

GradeUp sorts Unicode in a sort order determined by region settings in Windows. Upper and lower case values are considered equal.

GradeUp can not be used on boxed arrays.

GradeUp is a stable sort. It does not reorder elements which are equal.

Examples:

```
   |/: 3 1 2
 1 2 0
   1 2 0 <- 3 1 2
```

```
1 2 3
   |/: 1 2 0
2 0 1
   2 0 1 <- 1 2 3
3 1 2
```

The Monadic Iota Verb

The monadic Iota verb is a verb of rank 1 . It is applied to a scalar or vector right argument. When applied to a right argument of higher rank, it calls the monadic array operation helper program to loop over the parts of the argument it does not handle.

Monadic Iota is an index generator.

The result of monadic Iota is a sequence of natural numbers starting at 0.

With a scalar right argument the result is a vector of consecutive natural numbers. The length of the vector of natural numbers is the absolute value of the right argument. If the right argument is positive this vector is sorted in ascending order. If the right argument is negative the vector is sorted in descending order.

With a vector right argument the result is an array of natural numbers. The rank of the resulting array is the same as the length of the vector right argument. The shape of the resulting array is the same as the absolute value of the vector right argument. The resulting array contains all consecutive natural numbers from 0 to one less than the number of cells in the result array. If a cell in the vector right argument is positive the natural numbers along the corresponding dimension are sorted in ascending order. If a cell in the vector right argument is negative the natural numbers along the corresponding dimension are sorted in descending order.

Monadic Iota is used for indexing and sorting. One application is reversal of the sort order of dimensions in an array.

Examples:

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

The Open Monadic Verb

The Open monadic verb is a verb of rank 0. It is applied to the cells of the right argument. If the argument contains more than one cell, the monadic array operation helper program is used as a helper function.

The Open verb removes one layer of boxing of its right argument. It there are several layers of boxing, the result is still boxed. If not, the result is a noun of one of the non-boxed data types, the noun content of the boxed right argument.

If you open a noun which is not boxed the result is the same noun.

Open can be used in verb definition arguments to adverbs and conjunctions to open boxed arguments. You normally have to open boxed nouns with Open before you can operate on them. After doing your operation you can box the results.

Examples:

```
   |> |< 1 2 3
1 2 3
   |> ( |< 2 1 $ 0 1 ) , |< 1 3 $ 0 1 2
Type: Integer, Shape: 2 2 3 , Values: 0 0 0 1 0 0 0 1 2 0 0 0 ...
   |> '/ 0 / ( |< 2 1 $ 0 1 ) , |< 1 3 $ 0 1 2
Type: Integer, Shape: 2 2 3 , Values: 0 0 0 1 0 0 0 1 2 0 0 0 ...
   |> |< |< 1 2 3
1 2 3
   |$ |> |< |< 1 2 3

   |> 1 2 3
1 2 3
   |$ |> 1 2 3
3
   "" {! |< ( |> [ ) , |> ] } / "John " "Lennon"
John Lennon
```

The ShapeOf Monadic Verb

The ShapeOf monadic verb is a verb of rank _ . It is applied to the whole of the right argument. There is no underlying verb.

ShapeOf returns the shape of the right argument.

Examples:

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

The Tally Monadic Verb

The Tally monadic verb is a verb of rank _ . It is applied to the whole of the right argument. There is no underlying verb.

Tally returns the size of the right argument in its first dimension.

Examples:

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

```
3 4 5
   |# |i. 2 3
2
```

Non-Scalar Dyadic Verbs

The non-scalar dyadic verbs are shown in table XXX

Table 8. Non-Scalar Dyadic Verbs

>-	Amend
>.-	Drop
<-	From
i.	Dyadic Iota
#	Replicate
/::	Select
$	Shape
<.-	Take
\::	Transpose

The Amend Dyadic Verb

The Amend dyadic verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Amend is used to replace k-cells of the right argument. The result is a new noun where the k-cells are replaced. The original noun is not modified.

The left argument is a boxed array with two values. The first box contains indexes along the first dimension of the right argument. The second box contains the k-cells with which the present contents at these indexes should be replaced.

Example:

```
   ( ( |< 1 2 ) , |< -4 -5 ) >- 0 1 2 3 4
0 -4 -5 3 4
```

The Drop Dyadic Verb

The Drop dyadic verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Drop is used to remove k-cells of the right argument. The result is a new noun where these k-cells are removed. The original noun is not modified.

If the left argument is a positive scalar, it denotes the number of k-cells that should be removed from the head of the right argument.

If the left argument is a negative scalar, it denotes the number of k-cells that should be removed from the tail of the right argument.

If the left argument is a vector of length n, it denotes the number of k-cells that should be removed from the head or tail of the first n dimensions of the right argument.

If the number of k-cells to remove is larger than the array length in the corresponding dimension, all k-cells in this dimension are removed.

Example:

```
   2 >.- 0 1 2 3
2 3
   -2 >.- 0 1 2 3
0 1
   1 1 >.- |i. 2 2 2
6 7
```

The From Dyadic Verb

The From dyadic verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

From is used to select k-cells of the right argument. The result is a new noun with the selected k-cells. The original noun is not modified.

The left argument contains indexes along the first dimension of the right argument.

From can be used together with the dyadic Rank operator to select k-cells along other dimensions than the first dimension.

Example:

```
   1 2 <- 1 2 3 4
2 3
   1 2 <- '/ 1 1 / |i. 2 4
1 2
5 6
```

The Dyadic Iota Verb

The dyadic Iota verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. If the left argument is not a scalar, vector or table the result is rank error. There is no underlying verb. It does not handle boxed data.

The dyadic Iota verb searches in the k-cells of the left argument for corresponding k-cells in the right argument. If the k-cell of the right argument is found in the k-cells of the left argument the corresponding cell in the result contains the index of the first k-cell in the left argument equal to the right argument k-cell. If the k-cell of the right argument is not found in the k-cells of the left argument the corresponding cell in the result contains the length of the left argument.

Example:

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

```
   1 2 3 i. |i. 2 2
3 0
1 2
   ( |i. 2 2 ) i. |i. -4 2
2 2 1 0
```

The Replicate Dyadic Verb

The Replicate dyadic verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Replicate is used to replicate k-cells of the right argument. The result is a new noun with the replicated k-cells. The original noun is not modified.

The left argument contains an integer vector of the same length as the first dimension of the right argument.

The result contains the k-cells of the right argument, each replicated the number of times the corresponding left argument integer value shows.

If the corresponding integer in the left argument is 0, the corresponding k-cell in the right argument will not be part of the result.

Replicate can be used together with the dyadic Rank operator to replicate k-cells along other dimensions than the first dimension.

Example:

```
   0 1 2 0 # 1 2 3 4
2 3 3
   0 1 2 0 # '/ 1 1 / |i. 2 4
1 2 2
5 6 6
```

The Select Dyadic Verb

The Select dyadic verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Select is used to select k-cells of the right argument. The result is a new noun with the selected k-cells. The original noun is not modified.

The left argument contains a boolean vector of the same length as the first dimension of the right argument.

The result contains the k-cells of the right argument corresponding to true values in the left argument boolean vector.

Select can be used together with the dyadic Rank operator to select k-cells along other dimensions than the first dimension.

Example:

```
   false true true false /:: 1 2 3 4
2 3
   false true true false /:: '/ 1 1 / |i. 2 4
1 2
```

```
5 6
```

The Shape Dyadic Verb

The Shape dyadic verb is a verb of rank 1 _ . It is applied to the whole of the right and left argument. The dyadic Rank operator is used as a helper program to handle left arguments of rank higher than 1

The Shape dyadic verb reshapes the k-cells of the right argument to the shape defined by the value of the vector left argument.

The shape of the result is the shape of a k-cell appended to the value of the left argument.

If the right argument does not contain any values there will be an error message.

If the right argument contains fewer values than the result to be produced, values are taken like if there were an indefinite sequence of catenated right arguments to take from.

Example:

```
    2 2 $ |i. 0
Length error
    2 2 $ 1 2 3 4
 1 2
 3 4
    2 2 $ 1 2
 1 2
 1 2
    2 2 $ |i. 2 2
Type: Integer, Shape: 2 2 2 , Values: 0 1 2 3 0 1 2 3
```

The Take Dyadic Verb

The Take dyadic verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Take is used to take k-cells of the right argument. The result is a new noun of these k-cells. The original noun is not modified.

If the left argument is a positive scalar, it denotes the number of k-cells that should be taken from the head of the right argument.

If the left argument is a negative scalar, it denotes the number of k-cells that should be taken from the tail of the right argument.

If the left argument is a vector of length n, it denotes the number of k-cells that should be taken from the head or tail of the first n dimensions of the right argument.

If the number of values to take is larger than the array length in the corresponding dimension, k-cells with *fill values* are added.

The integer fill value is 0, the float fill value is 0.0, the boolean fill value is false, the unicode fill value is blank and the boxed array fill value is a scalar boxed array containing an empty integer vector.

Example:

```
   2 <.- 0 1 2 3
0 1
   -2 <.- 0 1 2 3
2 3
   1 1 <.- |i. 2 2 2
0 1
   6 <.- 0 1 2 3
0 1 2 3 0 0
```

The Transpose Dyadic Verb

The Transpose dyadic verb is a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Transpose is used to transpose the array of the right argument. The result is a new noun with the transposed array. The original noun is not modified.

The left argument is a vector of the same length as the rank of the right argument. The values of the left argument are the same as if they were used to index the shape of the right argument. If the rank of the right argument is three the left argument has to contain 0, 1 and 2.

The result is that the dimensions change order to the order described by the left argument index.

Example:

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

The JWithATwist Adverbs

The adverbs are shown in the following table:

Table 9. Adverbs

/	The Fold adverb
|'	The Monadic Rank adverb
'	The Dyadic Rank adverb
\	The Scan adverb

The Fold Adverb

The Fold adverb with its verb argument forms a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Fold applies the verb argument between an accumulator and successive k-cells of the right argument. The result is the accumulator value after the last verb application. The left value is the initial accumulator value.

The accumulator and all verb application results must have the same shape and type.

If the accumulator is boxed, it's contents can have any shape and type.

Example:

```
    0 + / 1 2 3
 6
```

The Monadic Rank Adverb

The monadic Rank adverb with its verb argument forms a verb of rank _ . It is applied to the whole of the right argument. There is no underlying verb.

The monadic Rank adverb applies its verb to k-cells of the right argument, parts of the right argument with a certain rank.

The monadic Rank adverb has a rank argument, evaluated during interpretation. The rank argument defines the k-cells of the verb application. If the rank argument is positive, it denotes the k-cell from the end of the shape of the argument. If the rank argument is negative, it denotes the k-cells from the beginning of the shape of the right argument.

If we have an argument of shape 2 3 4 5 a rank of 1 would mean we operate on the shape 5 k-cells, a rank of -1 would mean we operate on the shape 3 4 5 k-cells.

The part of the shape to the left of the part we operate on is called the outer shape.

The verb application will result in a number of nouns, possibly of different shape.

The Rank adverb will calculate the maximum length of these resulting nouns in each of their dimensions. The result is called the inner shape.

The Rank adverb will reshape each of the produced results to the rank of the inner shape by adding ones to the left.

The Rank adverb will then use the Take dyadic verb to create arrays with the shape of the inner shape from each of the produced results. In this process *fill values* are added.

The Rank adverb will then put the maximized results in the parts of the result array defined by the outer shape.

The result will have a shape defined by catenating the outer and the inner shape.

Example:

```
    |< |'/ 0 / 1 2 3
 1 2 3
    |$ |< |'/ 0 / 1 2 3
 3
    |< |'/ 1 / 1 2 3
1 2 3
    |$ |< |'/ 1 / 1 2 3

    {! |< 1 + ] } |'/ 0 / 1 2 3
```

```
 2 3 4
    |i. |'/ 1 / 2 2 $ 2 1 1 3
Type: Integer, Shape: 2 2 3 , Values: 0 0 0 1 0 0 0 1 2 0 0 0 ...
```

If we operate on an empty argument the monadic Rank adverb has special cases to handle. The argument as a whole can be empty because the shape contains zeroes, but the part of the argument we operate on can be non-empty. If there is nothing to apply the verb to the shape and type of the result can not be determined by normal execution. It is still very important to handle these cases. What JWithATwist does is to create a *fill value* object or fill value objects of the same shape and type as the non-empty part of the argument. The verb is applied to the fill value object. The resulting shape and type is used as the inner shape and the result type.

If the application of the verb on the fill value objects cause an error, the program terminates with this error.

If the verb application on the fill value objects have results of the correct shape and type, and does not terminate with an error, the designed program using the Rank adverb will work for the special empty argument cases.

The Dyadic Rank Adverb

The dyadic Rank adverb with its verb argument forms a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

The dyadic Rank adverb applies its verb to k-cells of the left and right arguments, parts of the left and right arguments with a certain rank.

The dyadic Rank adverb has a rank argument, evaluated during interpretation. The rank argument defines the k-cells of the verb application. If the rank argument is positive, it denotes the k-cell from the end of the shape of the argument. If the rank argument is negative, it denotes the k-cells from the beginning of the shape of the argument.

If we have an argument of shape 2 3 4 5 a rank of 1 would mean we operate on the shape 5 k-cells, a rank of -1 would mean we operate on the shape 3 4 5 k-cells.

The part of the shape to the left of the part we operate on is called the outer shape.

This outer shape must be the same for the left and right argument.

The verb application will result in a number of nouns, possibly of different shape.

The rank adverb will calculate the maximum length of these resulting nouns in each of their dimensions. The result is called the inner shape.

The Rank adverb will reshape each of the produced results to the rank of the inner shape by adding ones to the left.

The Rank adverb will then use the Take dyadic verb to create arrays with the shape of the inner shape from each of the produced results. In this process *fill values* are added.

The Rank adverb will then put the maximized results in the parts of the result array defined by the outer shape.

The result will have a shape defined by catenating the outer and the inner shape.

Example:

```
   1 2 <- '/ 1 1 / |i. 2 4
1 2
5 6
   10 20 {! |< [ + ] }  '/ 1 1 / |i. 2 2
10 21 12 23
   |$ 10 20 {! |< [ + ] }  '/ 1 1 / |i. 2 2
2
   ( 2 2 $ 10 20 30 40 ) {! |< [ + ] }  '/ 1 2 / |i. 2 2
10 11 30 31
22 23 42 43
   |$ ( 2 2 $ 10 20 30 40 ) {! |< [ + ] }  '/ 1 2 / |i. 2 2
2
    10 20 + |i. 2 2
10 11
22 23
```

If we operate on empty arguments the dyadic Rank adverb has special cases to handle. The argument or arguments as a whole can be empty because the shape contains zeroes, but the part of the argument we operate on can be non-empty. If there is nothing to apply the verb to the shape and type of the result can not be determined by normal execution. It is still very important to handle these cases. What JWithATwist does is to create a *fill value* object or fill value objects of the same shape and type as the non-empty parts of the argument or arguments. The verb is applied between one argument and the fill value object or between two fill value objects. The resulting shape and type is used as the inner shape and the result type.

If the application of the verb on the fill value objects cause an error, the program terminates with this error.

If the verb application on the fill value object or objects have results of the correct shape and type, and does not terminate with an error, the designed program using the Rank adverb will work for the special empty argument cases.

The Scan Adverb

The Scan adverb with its verb argument forms a verb of rank _ _ . It is applied to the whole of the right and left argument. There is no underlying verb.

Scan applies the verb argument between an accumulator and successive k-cells of the right argument. The result is an array of all accumulator values after the last verb application. The left value is the initial accumulator value.

The accumulator and all verb application results must have the same shape and type.

If the accumulator is boxed, it's contents can have any shape and type.

Example:

```
   0 + \ 1 2 3
1 2 6
```

The JWithATwist Conjunctions

For the moment JWithATwist has no conjunctions and this section is empty. Many of the operations of the J conjunctions can be done in other ways in JWithATwist.

The JWithATwist Syntax

These simple or simplified verbs, adverbs and conjunctions are used to make this description easier to understand:

- Dyadic verb Addition, `+`
- Dyadic verb Subtraction, `-`
- Monadic verb Negation, `|-`
- Monadic adverb, `/` . This operator is defined `u / y <-> y u y`. It applies the in-fix u verb between two copies of the argument. `+ / 1 <-> 1 + 1 <-> 2`.
- Dyadic adverb, `/-` . This operator is defined `x u/- y <-> x u y`. It applies the infix u verb between the left and right argument. `1 + /- 1 <-> 1 + 1 <-> 2`
- Monadic conjunction, `|.` . This operator is defined `u |. v y <-> u y v y`. It applies the infix dyadic verb v between copies of the right argument and the monadic verb u to the result. `|- |. + 1 <-> |- 1 + 1 <-> -2 .`
- Dyadic conjunction, `.` . This operator is defined `x u . v y <-> u x v y` . It applies the infix dyadic verb v between the left and right arguments and the monadic verb u to the result. `2 |- . + 5 <-> |- 2 + 5 <-> -7 .`

There is no order of precedence between the operations. Execution is from right to left modified by parenthesis. `|- |- |- 1 <-> -1`

The language elements are associated to its arguments like this:

- A monadic verb has one noun argument to the right. Like this: `|- 1` .
- A dyadic verb has one noun argument to the left and one to the right. Like this: `1 + 1` .
- A monadic adverb has a verb argument to the left and a noun argument to the right. Like this: `+ / 1` .
- A dyadic adverb has a verb argument to the left, one noun argument to the left and one to the right. Like this: `1 + /- 1` .
- A monadic conjunction has a verb argument to the left and one to the right, and one noun argument to the right. Like this: `|- |. + 1` .
- A dyadic conjunction has a verb argument to the left and one to the right, and one noun argument to the left and one to the right. Like this: `1 |- . + 1`.
- An adverb to the left of a conjunction is together with it's verb the left argument of the conjunction. Like this in the monadic case: `+ / |. + 1` . Like this in the dyadic case: `1 + / . + 1` .

An adverb together with its left verb forms a new verb. A conjunction together with its right verb and its left verb or adverb forms a new verb.

Monadic verbs, adverbs and conjunctions create verb trains like this:

```
|- + / |- |- |. + |- + / |. + |- 1
```

I use extra blanks to show the separation between verbs.

You always need at least one blank between two syntax elements.

A program is called a *noun definition*. It is written between curly brackets, like this `{ some code }` . It can contain *verb definitions*, written between curly brackets where the first bracket is immediately followed by `!` . Like this `{! some code }` .

A verb definition can contain Left Noun and Right Noun, denoted by `[` and `]` . The Left Noun is a placeholder for the noun immediately preceding the verb definition. The Right Noun is a placeholder for the noun immediately following the verb definition.

A verb definition containing only Right Nouns forms a monadic verb. A verb definition containing a Left Noun forms a dyadic verb, even if there is no Right Noun. In that case you are forced to have a noun to the right of the verb definition and this noun is silently ignored.

This is a verb definition of a monadic increment verb `{! 1 + ] }` . Example of use: `{! 1 + ] } 2` .

This is a verb definition of a dyadic addition verb `{! [ + ] }` . Example of use: `1 {! [ + ] } 2` .

Left and Right nouns within brackets are placeholders for the noun arguments to the first verb definition enclosing the brackets. In this example the Left noun is one and the Right noun is two: `1 {! ( [ + ] ) + 1 } 2` .

There is no way to define adverbs and conjunctions in this version of JWithATwist.

A sample session :

```
{ 1 }
 1
{ 1 + 1 }
 2
{ ( 1 - 1 ) + 1 }
 1
{ ( 1 - 1 ) {! [ + ] } 1 }
 1
{ { 1 - 1 } {! [ + ] } 1 }
 1
{ + / 1 }
 2
{ {! [ + ] } / 1 }
 2
{ + / |. + 1 }
 4
{ 2 + / . + 1 }
 6
{ 2 + / . + |- |- 1 }
 6
{ 2 + / . {! [ + ] } |- |- 1 }
 6
{ 2 + / . + |- {! |- ] } 1 }
 6
{ 2 + /- 3 }
 5
{ 5 {! ( [ + ] ) + [ + ] } 7 }
 24
{ |- + / |- |. + + / |. + |- 5 }
```

```
-80
```

JWithATwist Use Cases

JWithATwist can be used as a calculator in a terminal window with the included JWithATwist program. JWithATwist can also be used from F#. You can use the JWithATwist verbs, adverbs and conjunctions directly from F#, with no interpreter in between. You can define J verbs and use them as F# functions. You can use JWithATwist definitions as lambda expressions or unnamed variables and functions.

JWithATwist can easily be extended. If you implement a new verb, adverb or conjunction as an F# function it can be included in the interpreter with only some minutes of work. You can also easily extend JWithATwist with verbs you define from JWithATwist expressions.

The JWithATwist interpreter can be used separately. You can create your own language. The JWithATwist Mock Interpreter is a good starting point. If you want the basic APL/J syntax it is an easy way to create your own language.

The JWithATwist Terminal

Install JWithATwist with the included Windows installation program. You will get a JWithATwist program in your Start menu. Run it and start writing JWithATwist expressions. Remember that all expressions are written within curly brackets. Also remember to put blanks between all language elements. If you press return without any input you will terminate the terminal session.

If you cloned JWithATwist to your development environment you can start the terminal session by calling the Parser program.

Examples :

```
{ 1 }
 1
{ 1 + 1 }
 2
{ {! 1 + ] } |i. 3 }
 1 2 3
```

Using JWithATwist From F#

All WithATwist verbs, adverbs and conjunctions can be used directly from F#. Usually the name is the same as the name of the language element preceded by J. JOpen, JBox, JAdd and JSubtract are some examples. Sometimes there is a monadic and a dyadic version of the language element. Then Monadic or Dyadic is appended to the name. JIotaMonadic, JIotaDyadic, JRankMonadic and JRankDyadic are some examples.

When you use the verbs, adverbs and conjunctions you need noun arguments.

A noun is a simple record structure containing type, shape and an F# array of values. You can see the type definitions at the top of the JWithATwist.Base module. You can easily create a noun from F# variables.

If you want to create a noun from a JWithATwist expression you can use the JNounDefine helper function. It takes a string right argument with a JWithATwist expression and returns a noun.

If you want to use a resulting noun in F# you have to use the record structure. There is no helper function to create any special F# representation from a noun.

You can also create F# functions from JWithATwist expressions. There are two helper functions, JVerbMonadicDefine and JVerbDyadicDefine. They take a string right argument with a JWithATwist expression and return a monadic or dyadic verb.

Sometimes you want to create a verb directly in F# from other verbs. You can just create a function with one or two noun arguments and use these in the function definition.

You can also partially execute verbs, adverbs and conjunctions in F#, and create new verbs, adverbs or conjunctions as a result.

If you define the JWithATwist verbs at the module level, they will only be interpreted once, when the module is loaded. After definition JWithATwist verbs are a composition of compiled code. There is no interpretation when the resulting F# function is executed.

Examples :

```
    JPrint (JIotaMonadic (JNounDefine @"{ 3 }"))
 0 1 2
    let Increment = JVerbMonadicDefine @"{! 1 + ] }"
    JPrint (Increment (JIotaMonadic (JNounDefine @"{ 3 }")))
 1 2 3
    let Add = JVerbDyadicDefine @"{! [ + ] }"
    JPrint (Add (JNounDefine @"{ 1 }") (JNounDefine @"{ 1 }"))
 2
    let Increment = Add (JNounDefine @"{ 1 }")
    JPrint (Increment (JIotaMonadic (JNounDefine @"{ 3 }")))
 1 2 3
    let aNoun = JNounDefine @"{ 1 }"
    let f aNoun =
        match aNoun.JType,aNoun.JShape,aNoun.JValue with
        |aType,aShape,JTypeIntegerArray aValue ->
            printfn "%i" aValue.[0]
        |_->
            raise JExceptionSystemError
    f aNoun
1
```

The printout in the examples is what is written to standard output as a result of the preceding code sample.

Extending JWithATwist

To extend JWithATwist with new verbs, adverbs and conjunctions you first have to create the F# function. If you want you can use JWithATwist code in the definition. You can also use parsed JWithATwist expressions.

The extension is done in the ParserInterface module. You copy and modify the code for some similar language element. You just have to change some lines of code. Then you have to insert the new parsing code in the parser, by modifying parseVerbAdverbConjunction. All is very easy.

Using the JWithATwist Mock Interpreter to Create Your Own Language

If you want to create your own language and use the JWithATwist interpreter, you just copy the ParserMockDefinitions, ParserMockInterface and ParserMock files into your F# project. If you want you can

copy the test code in JWithATwist.Test.ParserMock. Run the Parser program in ParserMock and try the JWithATwist Mock Interpreter. Then you can change it by modifying the definitions in ParserMockDefinitions and the code in ParserMockInterface. You normally don't have to change the interpreter and the parse code in ParserMock. It is easy. Creating the language elements can be difficult, of course.

Glossary

Adverb	A function with a verb argument and one or two noun arguments, returning a noun.
Agreement	How JWithATwist allows operations between arguments of different shape.
Boxed	A noun of data type boxed contains other nouns. It is an n-dimensional array of n-dimensional arrays.
Cell	The smallest unit in an n-dimensional array. A simple data type or another noun.
Conjunction	A function with two verb arguments and one or two noun arguments, returning a noun.
Dyadic	A verb, adverb or conjunction with two noun arguments.
Fill value	Fill values can be used when a noun is resized. The integer fill value is 0, the float fill value is 0.0, the boolean fill value is false, the unicode fill value is blank and the boxed array fill value is a scalar boxed array containing an empty integer vector.
K-cell	The noun is seen as a structure of k-cells. The scalar is a 0-cell, the vector a 1-cell, the table a 2-cell and so on for higher dimensions.
Monadic	A verb, adverb or conjunction with one noun argument.
Noun	An n-dimensional array of values where the values are simple data types or other n-dimensional arrays.
Noun definition	A noun definition is written between curly brackets. It contains a noun or an expression returning a noun.
Rank	The rank of a noun is the number of dimensions of the array it contains. The rank of a verb is the size of k-cell or k-cells of the argument or arguments that the actual underlying verb operation is performed on.
Scalar	A noun with zero dimensions contains only one value. It is called a scalar. An operation on 0-cells is called a scalar operation.
Verb	A function with one or two noun arguments returning a noun.
Verb definition	A verb definition is written between curly brackets with the opening bracket followed by `!`. It an expression returning a verb.