ARR v1.0

*by **digi_cs** Sep-Nov 2024* *cosmogen@gmail.com*

Библиотека ARR предоставляет базовые функции для работы с массивами в AWK.

Библиотека включает в себя 25 функций для работы с массивами:

- 5 функций для объединения элементов массивов в строку.:

ret::a	вернуть элементы массива
ret::as	вернуть элементы массива с сепаратором
ret::ad	вернуть элементы массива
ret::ab	вернуть элементы двух массивов
ret::abs	вернуть элементы двух массивов с сепаратором

При создании этих функций использовались элементы технологий линейного программирования и поэтому они обладают высокой производительностью и расчитаны на большие объёмы данных и количества объединяемых индексов.

Смотрите раздел **ARRAY JOINING**

Смотрите раздел **PERFORMANCE**

- 3 функции для объявления субмассивов в массивах и удаления индексов из массивов:

def::ia	объявить чистый субмассив в массив[индекс]
let::ia	объявить субмассив в массив[индекс] если тот ещё не объявлен массивом
del::ia	удалить массив[индекс]

- 2 специальные функции

arr::same	два РАЗНЫХ массива ?
arr::name	определение имени глобального массива

- 2 функции для генерирования визуализации содержимого массивов

dump::a	дамп массива
dump::ab	дамп двух массивов

- 15 функций для копирования данных между массивами и их элементами:

let::aa **def::aa** **mov::aa**	массив1 < массив2
let::iav **def::iav** **mov::iav**	массив[индекс] < массив/значение
let::iai **def::iai** **mov::iai**	массив[индекс1] < массив[индекс2]
let::aia **def::aia** **mov::aia**	массив1[индекс] < массив2[индекс]
let::iaia **def::iaia** **mov::iaia**	массив1[индекс1] < массив2[индекс2]

Функции **let**, **def** и **mov** отличаются поведением при копировании массивов в массивы:

mov-функции просто копируют данные из источника в приёмник: при копировании массива в уже существующий массив данные в нём сохраняются за исключением тех которые будут переопределены индексами источника при копировании

def-функции всегда очищают таргет: массив или массив[индекс]: точное копирование

let-функции копируют только то, чего в приёмнике нет: т. е. копируются только те индексы источника которых нет в приёмнике

USE

Для использования выполните:

```
@include    "arr.lib"
```

Библиотека ..

FUNCSET

del::**ia**(*i*, *A*)

Delete *A*[*i*] (if exist)

Returns *i*.

let::**ia**(*i*, *A*)
def::**ia**(*i*, *A*)

Define *A*[*i*] as the subarray.

Returns *i*.

Specifuc:

def::ia always clear array *A*[*i*]

```
example:
```

```
… patsplit( ..., A[ def::a( "A", A ) ], …, A[ def::a( "B", A ) ] ) ...
```

patsplit will be targeted to arrays: A[“A”] and A[“B”]

let::**aa**(*D*, *S*)
def::**aa**(*D*, *S*)
mov::**aa**(*D*, *S*)

Copy all indexes from the source array *S* to destination array *D*.

if *S* and *D* is the same array then do nothing.

Returns null

Specifuc:

def::aa pre-clear array D

let::aa indexes of *S* are will be copied in case if it's not exist in *D*.

```
example:

      zz
```

let::iav(i, A, V)
def::**iav**(i, A, V)
mov::**iav**(i, A, V)

Copy V to A[i]

Returns i

V may be any of type (array or other)

Specific:

def::iav A[i] will be always deleted before V copied

def::iav if V is untyped then A[i] will be deleted

example:

```
zz
```

let::iai(d, A, s)
def::iai(d, A, s)
mov::**iai**(d, A, s)

zz

example:

zz

let::aia(*D, i, S)*
def::aia(*D, i, S)*
mov::**aia**(*D, i, S)*

zz

example:

zz

let::iaia(*d, D, s, S*)
def::iaia(*d, D, s, S*)
mov::**iaia**(*d, D, s, S*)

zz

example:

```
zz
```

arr::**same**(*A*, *B*)

Returns true (1) in case if the arrays *A* and *B* are the same array. Otherwise returns 0.

arr::**name**(*A*)

Returns the name of the given global array *A*.

If *A* is not global array then returns null.

dump::**a**(*A*, *name*)

Returns dump of the given array *A*

```
example:

      zz
```

dump::**ab**(*A*, *B*)

Returns dump of the two given arrays *A* and *B*

```
example:

      zz
```

ARRAY JOINING

Данная библиотека включает в себя реализацию четырёх операций связанных с объединением элементов массива(ов) в строку:

```
RETA        объединение элементов массива

            A[a]  A[a+=x]  A[a+=x]

RETAS       объединение элементов массива через сепаратор

            A[a]  sep  A[a+=x]  sep  A[a+=x]

RETAB       объединение пар элементов двух массивов

            A[a]B[b]  A[a+=x]B[b+=y]  A[a+=x]B[b+=y]

RETABS      объединение пар элементов двух массивов через сепаратор

            A[a]B[b]  sep  A[a+=x]B[ b+=y]  sep  A[a+=x]B[b+=y]
```

В процессе объединения элементов в строку — данные из объединяемых элементов считываются и копируются в результирующую строку. Это процесс называется data pass.

data pass — это когда некоторые данные считываются и копируются куда-то ещё. Именно этот процесс и является той самой нагрузкой потребляющий перформанс при джойнинге элементов массива .

При создании данной библиотеки первичной целью было минимизировать количество data pass для одних и тех-же данных. Количество необходимых data pass напрямую зависит от количества объединяемых индексов.

Ниже приведена таблица информирующая о количестве data pass для различного количества объединяемых индексов:

indexes	data pass
1 .. 64	1
65 .. 64K	2
64K .. 4M	3
4M .. 256M	4
256M .. 16G	5

1 - 64 индекса возвращаются за один data pass — т. е. напрямую из исходных массивов

65 - 64K объединяемых индекса возвращаются за два data pass: данные копируются из исходных массивов во временный массив и затем возвращаются из него

64K - 4M индексов возвращаются в 3 data pass: данные копируются из исходных массивов во временный массив, временный массив сжимается в 64 раза (AIR) и затем данные из него возвращаются

свыше 4M и до 256M объединяемых индексов возвращаются в 4 data pass: данные копируются из исходных массивов во временный массив, временный массив дважды сжимается в 64 раза (AIR) и затем данные из него возвращаются

свыше 256M и до 16G (миллиардов) индексов потребуют 5 data pass: : данные копируются из исходных массивов во временный массив, временный массив трижды сжимается в 64 раза (AIR) и затем данные из него возвращаются

операция RETA обладает двукратным количеством индексов для каждого data pass — т.е она возвращает до 128 индексов в один data pass и до 128K в два datapass и т.д.

ret::**a**(*A, a, q, sa, sep*)
ret::**ad**(*A, sa, a, q, sep*)
ret::**as**(*A, sep, a, q, sa*)

All three functions are doing the same things and differs from each other only by parameters order.

Returns specified number of indexes of an array with separator string between each index:

```
return A[a]sep A[a+=sa]sep … A[a+=sa]
```

A is the source array

the first index in array is calculated by the following

```
first =  ! ( 0 in A )
```

first index is equals to 0 in case if index 0 is exist in array *A*
otherwise first index equals to 1

last index in an array is calculated by the following:

```
last = first + length( A ) - 1
```

a is the start index in array *A*

if *a* is omitted (== "") then it's will be calculated depending from *sa* parameter (see below):

if index modifier (*sa*) is equals to zero or positive number (>= 0) then start index *a* will be first index in array *A*

otherwise if index modifier (*sa*) is less than zero (< 0) then start index *a* will be last index in array *A*

sa is the index modifier for an array *A*

if *sa* is omitted (== "") then default value 1 will be used

q is the number of indexes to return

if *q* is omitted (== "") then it's will be calculated for covering rest of the elements of the array *A* starting from determinated start index *a* and using determinated index modifier *sa*

sep is the separator string that will be placed between the content of the each index joined

Below is the examples of usage functions:

example 1:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::a( T )             # joining all elements of the array T

# t == t2 == "ABCDE"
```

example 2:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::ad( T, -1 )        # joining all elements of the array T
                             # in reverse order
# t  == "ABCDE"
# t2 == "EDCBA"
```

example 3:

```
split( t, L, /\n/ )          # splitting multilined t for each line

t2 = ret::as( L, "\n" )      # joining all elements (lines of t) of
                             # the array L with separator: "\n" (eol)
# t == t2
```

example 4:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::a( T, 3 )          # joining rest of the elements of the
                             # array T starting from index 3 using
                             # default index modifier (+1) with no
                             # separator
# t2 == "CDE"
```

example 5:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::ad( T, -1, 3 )     # joining rest of the elements of the
                             # array T starting from index 3 and using
                             # index modifier -1 with no separator
# t2 == "CBA"
```

ret::**ab**(*A, B, a, q, b, sa, sb*, *sep*)
ret::**abs**(*A, B, sep, a, q, b, sa, sb*)

Both functions are doing the same things and differs from each other only by parameters order.

Returns specified number of index-pairs of the two arrays with separator string between each pair:

```
return A[a]B[b]sep A[a+=sa]B[b+=sb]sep … A[a+=sa]B[b+=sb]
```

A, B are the source arrays

the first index in array is calculated by the following

```
first =  ! ( 0 in A/B )
```

first index is equals to 0 in case if index 0 is exist in array *A*/*B*
otherwise first index equals to 1

last index in an array is calculated by the following:

```
last = first + length( A/B ) - 1
```

a, *b* are the start indexes of the array *A*, *B* respectively

if *a*/*b* is omitted (== “”) then it's will be calculated depending from *sa*/*sb* parameter (see below):

if index modifier is equals to zero or positive number (>= 0) then start index *a*/*b* will be first index in an array *A*/*B*

otherwise if index modifier is less than zero (< 0) then start index *a/b* will be last index in an array *A*/*B*

sa, *sb* are the index modifiers for array *A*, *B* respectively

if *sa*/*sb* is omitted (== “”) then default value 1 will be used

q is the number of index-pairs to return

if *q* is omitted (== “”) then it's will be calculated for covering rest of the elements of the array *A* starting from determinated start index *a* and using determinated index modifier *sa*

sep is the separator string that will be placed between the content of the each index-pair joined

Below is the example of usage functions:

example:

```
f = "somefunc"

if ( q = patsplit( t, T, /.../, D ) ) {

      for ( t = 1; t <= q; t++ )

            T[ t ] = @f( T[ t ] )

      t = ret::ab( T, D ) }
```

В приведённом примере показан фрагмент типичного процессинга указателей в строке: мы разбиваем исходную строку в два массива: некие указатели и данные между ними, затем мы вызываем некоторую функцию для каждого из найденых указателей, а возвращаемый каждый раз результат этой функции сохраняем в том-же массиве вместо тела найденного указателя. Когда мы «пробегаем» все найденые указатели мы **собираем строку обратно с помощью объединения двух массивов в строку функцией ret::ab**

ARRAYS IN AWK

to define names (A,B,C,...) as the arrays use:

“” in A in B in C in ...

simpliest data accummulation in an array A is possible using

A[length(A)] = new added data

please note that A MUST to be defined as an array before

to pass undefined subarray A[i] use the def::ia/let::ia functions:

A[def::ia(i , A)]

ret::ab() function is designed as the opposite to built-in split/patsplit functions called with the two target arrays specified

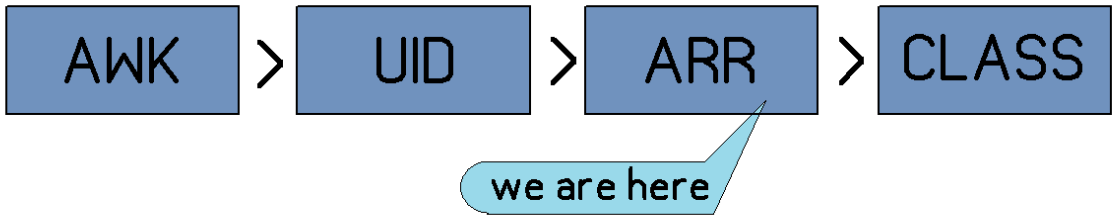
PERFORMANCE

FYI

PROSPECTS

The next release will be the CLASS – the OOP implementation for AWK: the fundamentals of basic techniques and a description of OOP practices in AWK.

A hypothetical publication plan for AWK libraries for the years 2024-2025:



AUTHOR

RESPECT DUE