Burleseque - Moonpage

Roman Müntener

Contents

ABOUT

An interpreter for the esoteric programming language *The Burlesque Programming Language*. Actually, Burlesque is less of a programming language than it is a tool. The actual language behind it is very simple and the only thing that makes Burlesque notable is the amount of built-ins it has. The syntax can be learnt within a few minutes (there are basically only Numbers, Strings and Blocks) and the concepts can be learnt quickly as well. People familiar with functional programming languages will already know these concepts so Burlesque is especially easy to learn if you already know the terms *map*, *filter*, *reduce*, *zip* and others. This moonpage tries to be as accurate, complete and easy to understand as possible. If you encounter an error in the documentation please report it on github. **Author:** Roman Müntener, 2012-?

Useful Weblinks:

- Burlesque on RosettaCode
- Source code
- Language Reference

Until this moonpage is complete, please consult the Language Reference. Once compelete, the moonpage will superseed the Language Reference.

HISTORY

Burlesque has been under development since 2012 and is still being improved on a regular basis. It was built as a tool for me (mroman) to use as a helper for my computer science studies.

SYNOPSIS

```
blsq <options>
```

LANGUAGE

SYNTAX

BUILT-INS

Add .+

`Int a, Int b:` Integer addition.

```
blsq ) 5 5.+
10
```

`Double a, Double b:` Double addition.

```
blsq ) 5.1 0.9.+
6.0
```

`String a, String b:` Concatenates two strings.

```
blsq ) "ab" "cd" .+
"abcd"
```

`Int a, String b:` Returns the first `a` characters of `b` as a String.

```
blsq ) 3 "abcdef" .+
"abc"
```

`Block a, Block b:` Concatenates two blocks.

```
blsq ) {1 2}{3 4}.+
{1 2 3 4}
```

`Char a, Char b:` Creates a string with the two characters `a` and `b` in it (in that exact order).

```
blsq ) 'a'b.+
"ab"
```

`String a, Char b:` Append `b` to `a`.

```
blsq ) "ab"'c.+
"abc"
```

`Int a, Block b:` Returns the first `a` elements of `b`.

```
blsq ) 2{1 2 3}.+
{1 2}
```

`Block a, Int b:` Returns the first `b` elements of `a`.

```
blsq ) {1 2 3}2.+
{1 2}
```

`String a, Int b:` Returns the first `b` characters of `a` as a String.

```
blsq ) "abc"2.+
"ab"
```

`Double a, Int b:` Convert `b` to Double, then perform addition.

```
blsq ) 1.0 2.+
3.0
```

`Int a, Double b:` Convert `a` to Double, then perform addition.

```
blsq ) 2 1.0.+
3.0
```

AddX _+

`Int a, Int b:` Creates a Block with the two Integers `a` and `b` as elements (in this exact order).

```
blsq ) 1 2_+
{1 2}
```

`Double a, Double b:` Creates a Block with the two Doubles `a` and `b` as elements (in this exact order).

```
blsq ) 1.0 2.0_+
{1.0 2.0}
```

`String a, String b:` Concatenates the two Strings.

```
blsq ) "ab""cd"_+
"abcd"
```

`Block a, Block b:` Concatenates the two Blocks.

```
blsq ) {1}{2}_+
{1 2}
```

`Char a, Char b:` Converts both arguments two string and concatenates them.

```
blsq ) 'a'b_+
"ab"
```

`String a, Char b:` Converts `b` to String, then concatenates.

```
blsq ) "a"'b_+
"ab"
```

`Char a, String b:` Converts `a` to String, then appends it to `b`.

```
blsq ) 'a"b"_+
"ba"
```

`Int a, String b:` Converts `a` to String, then appends it to `b`.

```
blsq ) 1"b"_+
"b1"
```

`String a, Int b:` Converts `b` to String, then concatenates.

```
blsq ) "b"1_+
"b1"
```

Append `[+`

`Block a, Any b:` Append `b` to `a`.

```
blsq ) {1 2}9[+
{1 2 9}
```

`String a, Char b:` Append `b` to `a`.

```
blsq ) "ab"'c[+
"abc"
```

`Int a, Int b:` Concatenates Integers.

```
blsq ) 12 23[+
1223
```

Authors' Notes: This built-in is rather superfluous because most of its use-cases can be covered by either using `++` or `_+`. Yet, there may be some rare use-cases where you might want to use it for example in `{[+}r[` or the like.

Average `av`

`Block a:` *Defined as:* *`J++jL[pd./`*. Calculates average.

```
blsq ) {1 2 3 4}J++jL[pd./
2.5
blsq ) {1 2 3 4}av
2.5
```

Authors' Notes: The `pd` is there to ensure that the result is always a Double.

`Block a:` Floor of `a` as Integer.

```
blsq ) 5.9av
5
blsq ) 5.1av
5
```

Authors' Notes: If you want the floor of `a` as Double use `fo`.

Average2 `AV`

Defined as: `PDav`. Calculates average.

```
blsq ) {1 2 3.2 4}AV
2.75
blsq ) {1 2 3.2 4}av
2.55
```

Authors' Notes: This built-in is a relic from earlier versions of Burlesque where `./` only worked on `Int, Int` or `Double, Double` but not with `Double, Int` or `Int, Double`. This meant that `av` could only be used on Blocks that contained Doubles (because `++` would produce an Integer otherwise and the `L[pd./` would fail because an Integer could not be divided by a Double). In such cases `AV` had to be used. With newer versions the functionality of `./` was extended and `av` can now be used on Blocks that contain Integers as well. However, if you use `AV` on a Block that contains Doubles it will convert all these Doubles to ceiling(a) which is not what you want in most cases. Thus: The use of `av` is recommended as it is safer.

Concat `\[`

`Block {}`: *Defined as:* `{}`. Empty Block becomes empty Block.

```
blsq ) {}\[
{}
```

`Block {Block (Char a)}`: Return a single character string consisting of `a`.

```
blsq ) {'a}\[
"a"
```

`Block a`: *Defined as:* `{_+}r[`. Concatenates elements in a Block.

```
blsq ) {{1 1} {2 1} {3}}{_+}r[
{1 1 2 1 3}
blsq ) {{1 1} {2 1} {3}}\[
{1 1 2 1 3}
blsq ) {'a 'b 'c}\[
"abc"
```

Authors' Notes: There is an additional special case when `{_+}r[` does not return a Block the return value will be boxed. Why this special case exist remains unknown.

ConcatMap `\m`

Defined as: `m[\[`.

```
blsq ) {1 2 3}{ro}m[\[
{1 1 2 1 2 3}
blsq ) {1 2 3}{ro}\m
{1 1 2 1 2 3}
blsq ) {1 2 3}{ro}m[
{{1} {1 2} {1 2 3}}
blsq ) "abc"{'ajr@}\m
"aababc"
```

Authors' Notes: The Map built-in detects if the input argument is a String and will concat automatically. Compare these examples:

```
blsq ) "abc"{'ajr@}m[
{'a 'a 'b 'a 'b 'c}
blsq ) "abc"XX{'ajr@}m[
{{'a} {'a 'b} {'a 'b 'c}}
blsq ) "abc"XX{'ajr@}\m
{'a 'a 'b 'a 'b 'c}
blsq ) "abc"XX{'ajr@}\m\[
"aababc"
blsq ) "abc"{'ajr@}\m
"aababc"
```

Continuation `c!`

Generally speaking a *Continuation* refers to executing code on a snapshot of the stack and then pushing the result back to the actual stack. This means that this built-in lets you run code without destroying data on the stack.

`Block a:` Run `a` as a Continuation.

```
blsq ) 5 4.+
9
blsq ) 5 4{.+}c!
9
4
5
blsq ) 5 4{.+J}c!
9
4
5
```

Decrement -.

`Int a`: Decrements `a`.

```
blsq ) 5-.
4
```

`Char a`: Returns the previous character (unicode point - 1)

```
blsq ) 'c-.
'b
```

`String a`: Prepend first character of `a` to `a`.

```
blsq ) "abc"-.
"aabc"
```

`Block a`: Prepend first element of `a` to `a`.

```
blsq ) {1 2 3}-.
{1 1 2 3}
```

Div ./

`Int a, Int b`: Integer division.

```
blsq ) 10 3./
3
```

`Double a, Double b`: Double division.

```
blsq ) 10.0 3.0./
3.3333333333333335
```

`String a, String b`: Removes `b` from the beginning of `a` iff `b` is a prefix of `a`.

```
blsq ) "README.md" "README" ./
".md"
blsq ) "README.md" "REDME" ./
"README.md"
```

`Block a, Block b`: Removes `b` from the beginning of `a` iff `b` is a prefix of `a`.

```
blsq ) {1 2 3}{1 2}./
{3}
blsq ) {1 2 3}{2 2}./
{1 2 3}
```

`Int a, Double b:` Converts `a` to Double, then divides.

```
blsq ) 10 3.0./
3.3333333333333335
```

`Double a, Int b:` Converts `b` to Double, then divides.

```
blsq ) 10.0 3./
3.3333333333333335
```

Duplicate `J ^^`

Duplicates the top most element.

```
blsq ) 5
5
blsq ) 5J
5
5
```

Equal `==`

`Any a, Any b:` Returns 1 if `a == b` else returns 0.

```
blsq ) 5 5==
1
blsq ) 5.0 5==
0
blsq ) 3 2==
0
blsq ) {1 23}{1 23}==
1
```

Eval `e!`

`Block a:` Evaluates (executes) `a`.

```
blsq ) {5 5.+}e!
10
```

Authors' Notes: If you want to eval a String use `pe`.

EvalMany `E!`

Defined as: `.*\[e!`. This built-in can be used to evaluate a Block a number of times.

```
blsq ) 1{J.+}1E!
2
blsq ) 1{J.+}2E!
4
blsq ) 1{J.+}3E!
8
blsq ) 1{J.+}4E!
16
blsq ) 1{J.+}4.*\[e!
16
```

Greater `.>`

`Any a, Any b:` Returns 1 if `a > b` else returns 0.

```
blsq ) 3.0 2.9 .>
1
blsq ) 2.0 2.9 .>
0
blsq ) 10 5 .>
1
blsq ) 10 5.0 .>
0
blsq ) 'a 1 .>
1
blsq ) 'a 9.0 .>
1
blsq ) 'a {} .>
0
blsq ) {} 9.0 .>
1
blsq ) {} 9 .>
1
```

Note: Comparing values with different types may result in unexpected (but determinstic, thus not undefined) behaviour.

Head `-]`

`Block a:` Returns the first element of `a`.

```
blsq ) {2 4 0}-]
2
```

`String a:` Returns the first character of `a`.

```
blsq ) "abc"-]
'a
```

Authors' Notes: If you need the first character of `a` as a String use `-~`.

`Int a:` Returns the first digit of `a` (works on absolute value).

```
blsq ) -451-]
4
```

HeadTail `-~`

Defined as: `-][-`.

```
blsq ) "abcd"-][-
"a"
blsq ) {{1 2 3} {4 5 6}}-~
{2 3}
blsq ) "abcd"-~
"a"
```

Authors' Notes: Useful to get the first character of a String as a String.

IfElse `ie`

`Block a, Block b, Int a:` Executes `a` if b is not zero, otherwise executes `b`.

```
blsq ) 5{3.*}{2.*}1ie
15
blsq ) 5{3.*}{2.*}0ie
10
```

Authors' Notes: This built-in is terrible because in most real-world cases it requires at least two additional swaps to get the result of a predicate to the top of the stack.

Iff `if`

`Int a, Block b:` Executes `b` only iff `a` is not zero.

```
blsq ) 5 1{3.*}if
15
blsq ) 5 0{3.*}if
5
```

`Block a, Int b:` Executes `a` only iff `b` is not zero.

```
blsq ) 5{3.*}0if
5
blsq ) 5{3.*}1if
15
```

Increment `+.`

`Int a:` Increments a.

```
blsq ) 5+.
6
```

`Char a:` Returns the next character (unicode point + 1).

```
blsq ) 'a+.
'b
```

`String a:` Appends the last character of `a` to `a`.

```
blsq ) "abc"+.
"abcc"
```

`Block a:` Appends the last element of `a` to `a`.

```
blsq ) {1 2 3}+.
{1 2 3 3}
```

Init ~]

`Block a:` Returns all but the last elements of `a`.

```
blsq ) {1 2 3}~]
{1 2}
```

`String a:` Returns all but the last character of `a`.

```
blsq ) "12a"~]
"12"
```

`Int a:` Returns all but the last digit of `a` (as Integer) (works on absolute value).

```
blsq ) 451~]
45
```

InitTail ~-

Defined as: ~][-. Can be used to remove the first and last element of a String/Block.

```
blsq ) "abcd"~][-
"bc"
blsq ) "abcd"~-
"bc"
```

Intersperse [[

`Any a, Block b:` Inserts `a` between elements in `b`.

```
blsq ) 0{1 2 3}[[
{1 0 2 0 3}
```

`Char a, String b:` Inserts `a` between characters in `b`.

```
blsq ) 'x"abc"[[
"axbxc"
```

See also: *Intercalate*.

Last `[~`

`String a:` Returns the last character of `a`.

```
blsq ) "abc"[~
'c
```

`Block a:` Returns the last element of `a`.

```
blsq ) {1 2 3}[~
3
```

`Int a:` Returns the last digit of `a` (works on absolute value).

```
blsq ) 451[~
1
```

Lines `ln`

`String a:` Split `a` into lines.

```
blsq ) "abc\ndef\ngeh"ln
{"abc" "def" "geh"}
```

`Int a:` Number of digits in `a` (works on absolute value).

```
blsq ) 123ln
3
blsq ) -123ln
3
```

`Block a, Block b:` Returns whichever is longer. If both are equal in length `b` is returned.

```
blsq ) {1 2}{1 2 3}ln
{1 2 3}
blsq ) {1 2 4}{1 2 3}ln
{1 2 3}
```

See also: *WithLines*.

Map `m[`

`String a, Block f:` *Defined as: `jXXjm[\[`.* Applies `f` to every character in `a`.

```
blsq ) "aBc"{<-}jXXjm[\[
"AbC"
blsq ) "aBc"{<-}m[
"AbC"
```

`Block a, Block f:` Applies `f` to every element in `a`.

```
blsq ) {1 2 3 4 5}{J.*}m[
{1 4 9 16 25}
```

See also: *ConcatMap* and there are many other different versions and shortcuts for *Map*.

Max `>.`

`Any a, Any b:` Returns whichever is greatest.

```
blsq ) 5 6>.
6
blsq ) 6 5>.
6
blsq ) {12}12>.
{12}
```

Maximum `>]`

`Block a:` Returns the maximum of `a`.

```
blsq ) {1 2 3 2 1}>]
3
```

`String a:` Returns the maximum of `a`.

```
blsq ) "debca">]
'e
```

`Int a:` Returns the largest digit as an Integer.

```
blsq ) 1971>]
9
blsq ) 1671>]
7
```

Min `<.`

`Any a, Any b:` Returns whichever is smallest.

```
blsq ) 5 4<.
4
blsq ) 5 4<.
4
blsq ) 10 10.0<.
10
```

Minimum `<]`

`Block a:` Returns the minimum of `a`.

```
blsq ) {1 2 0 3}<]
0
```

`String a:` Returns the minimum of `a`.

```
blsq ) "bac"<]
'a
```

`Int a:` Returns the smallest digit as an Integer.

```
blsq ) 109<]
0
```

Mod `.%`

This is an auto-zip and auto-map built-in.

`Int a, Int b:` Integer modulo.

```
blsq ) 10 3.%
1
```

Mul `.*`

`Int a, Int b:` Integer multiplication.

```
blsq ) 2 3.*
6
```

`Double a, Double b`: Double multiplication.

```
blsq ) 2.0 3.0.*
6.0
```

`String a, Int b`: Creates a Block containing `a` exactly `b` times.

```
blsq ) "ab"3.*
{"ab" "ab" "ab"}
```

`Char a, Int b`: Creates a String containing `a` exactly `b` times.

```
blsq ) 'a 3.*
"aaa"
```

`Block a, Int b`: Creates a Block containing `a` exactly `b` times.

```
blsq ) {1 2}3.*
{{1 2} {1 2} {1 2}}
```

`String a, String b`: Appends `a` to `b` then reverses.

```
blsq ) "123""456".*
"321654"
```

`Int a, Double b`: Converts `a` to Double, then multiplies.

```
blsq ) 2 3.0.*
6.0
```

`Double a, Int b`: Converts `b` to Double, then multiplies.

```
blsq ) 2.0 3.*
6.0
```

NotEqual `!=`

Defined as: `==n!`.

```
blsq ) 4 4==n!
0
blsq ) 4 3==n!
1
blsq ) 3 4==n!
1
blsq ) 3 4!=
1
blsq ) 4 4!=
0
```

Parse `ps`

This built-in auto-maps if the argument given is a block.

`String a:` Tries to parse `a` with the Burlesque parser. (Tries to parse `a` as Burlesque code). Returns a Block.

```
blsq ) "5"ps
{5}
blsq ) "5 3.0.+"ps
{5 3.0 .+}
blsq ) "{5 3.0.+}m["ps
{{5 3.0 .+} m[}
```

Authors' Notes: This built-in is handy. Instead of doing something like:

```
blsq ) "5 3 6 7"wdri++
21
```

you can just do:

```
blsq ) "5 3 6 7"ps++
21
```

Pow `**`

`Int a, Int b:` Returns `a` to the power of `b` (`a ^ b`).

```
blsq ) 2 3**
8
```

`Double a, Double b:` Returns `a` to the power of `b` (`a ^ b`).

```
blsq ) 4.0 3.0**
64.0
```

`Block a, Block b:` Merges `a` and `b`. `c = a_1, b_1, a_2, b_2`

```
blsq ) {1 2 3}{4 5 6}**
{1 4 2 5 3 6}
```

`String a, String b:` Merges `a` and `b`.

```
blsq ) "123""456"**
"142536"
```

`Char a:` Returns the unicode codepoint of `a` as an Integer.

```
blsq ) 'A**
65
blsq ) 'a**
97
```

Prepend +]

`Block a, Any b:` Prepend `b` to `a`.

```
blsq ) {1 2}3+]
{3 1 2}
```

`String a, Char b:` Prepend `b` to `a`.

```
blsq ) "ab"'c+]
"cab"
```

`Int a, Int b:` Prepends `b` to `a` (result is an Integer).

```
blsq ) 12 23+]
2312
```

Product `pd`

`Block {}`: *Defined as: `1`*. The product of an empty block is one.

`Block a`: *Defined as: `{.*}r[`*. Calculates the product of a Block.

```
blsq ) {1 2 3 4}{.*}r[
24
blsq ) {1 2 3 4}pd
24
blsq ) {1 2 3.0 4}pd
24.0
```

`Int a`: *Defined as: `Shrd`*. Converts to double.

```
blsq ) 5Shrd
5.0
blsq ) 5pd
5.0
blsq ) 5rd
5.0
```

`Double a`: Ceiling of `a` as Integer.

```
blsq ) 1.1pd
2
```

Authors' Notes: If you want ceiling of `a` as Double use `cl`.

ProductMany `PD`

Defined as: `{pd}m[`. Just maps `pd` over a Block.

```
blsq ) {{1 2 3} 5 {2 4}}{pd}m[
{6 5.0 8}
blsq ) {{1 2 3} 5 {2 4}}PD
{6 5.0 8}
```

Authors' Notes: Can be used as a shortcut for `)pd`. Otherwise this built-in doesn't offer too much over `rd` as `rd` auto-maps.

ReadArray `ra`

This built-in auto-maps if the argument given is a Block.

`String a:` Parses an array in `[,]`-notation.

```
blsq ) "[1,2,3]"ra
{1 2 3}
blsq ) "[1,[2,4],3]"ra
{1 {2 4} 3}
blsq ) "[1,[2 4],3]"ra
{1 {2 4} 3}
blsq ) "[1,[2 4],,,,3]"ra
{1 {2 4} 3}
```

It should be noted that `,` are optional and multiple `,` will be skipped as well. Nesting is supported.

`Char a:` Returns 1 iff `a` is space, else returns 0.

```
blsq ) " \t\ra0"{ra}m[
{1 1 1 0 0}
```

ReadDouble `rd`

This built-in auto-maps if the argument given is a Block.

`String a:` Converts `a` to Double.

```
blsq ) "3.0"rd
3.0
```

`Int a:` *Defined as:* ***`pd`***.

`Double a:` No operation.

```
blsq ) 3.1rd
3.1
```

`Char a:` Returns 1 iff `a` is alpha, else returns 0.

```
blsq ) 'ard
1
blsq ) '1rd
0
```

Authors' Notes: This built-in is useful to convert every element in a Block to a Double:

```
blsq ) {3.0 5 "3.14"}rd
{3.0 5.0 3.14}
```

ReadInt `ri`

This built-in auto-maps if the argument given is a Block.

`String a:` Converts `a` to Int.

```
blsq ) "100"ri
100
blsq ) "-101"ri
-101
```

`Int a:` No operation.

```
blsq ) 5ri
5
blsq ) -5ri
-5
```

`Double a:` *Defined as:* `av`.

`Char a:` Returns 1 iff `a` is alpha numeric, else returns 0.

```
blsq ) 'ari
1
blsq ) '1ri
1
blsq ) '.ri
0
```

Authors' Notes: This built-in is useful to convert every element in a Block to an Integer:

```
blsq ) {"12" 12.0 13 12.7}ri
{12 12 13 12}
```

However, Doubles are not rounded to the nearest Integer but are truncated. Rounding everything to the nearest Integer can be done with for example `rd)R_`.

Reduce `r[`

`Block a, Block f:` Takes the first element of `a` and the second element of `a`, applies `f`, takes the next element of `a` and applies `f` again and continues like that. More symbolically speaking `{1 2 3 4}{.+}r[` becomes `1 2 .+ 3 .+ 4 .+`, `{1 2 3 5}{?-}r[` becomes `1 2 ?- 3 ?- 4?-` and so forth.

```
blsq ) {1 2 3 4}{.+}r[
10
blsq ) {1 2 3 4}{.*}r[
24
```

Reverse `<-`

`String a:` Reverses `a`.

```
blsq ) "123"<-
"321"
```

`Block a:` Reverses `a`.

```
blsq ) {4 5 6}<-
{6 5 4}
```

`Int a:` Reverses the digits of an Integer. (Works on absolute value).

```
blsq ) -123<-
321
```

`Char a:` Inverts case.

```
blsq ) 'a<-
'A
blsq ) 'B<-
'b
```

Round `r_`

This built-in accepts a Block as first argument, in which case an auto-map is performed.

`Double a, Int b:` Rounds `a` to `b` decimal points.

```
blsq ) 3.12 2r_
3.12
blsq ) 3.19 2r_
3.19
blsq ) 3.5 0r_
4.0
blsq ) {3.5 3.4}0r_
{4.0 3.0}
```

Round2 `R_`

Defined as: *`0r_pd`*.

```
blsq ) {3.5 3.4}0r_pd
12.0
blsq ) {3.5 3.4}R_
12.0
blsq ) 5.5R_
6
blsq ) 5.3R_
5
blsq ) 5.3 0r_pd
5
```

Authors' Notes: Even though `r_` can auto-map this built-in won't do the same *expected* job because `pd` will calculate the product of a Block. You may however use this fact as a shortcut for example for `{0r_}m[pd`. If you want to round every Double to the nearest Integer in a Block use `)R_`.

Smaller `.<`

`Any a, Any b:` Returns 1 if `a < b` else returns 0.

```
blsq ) 2 3.<
1
blsq ) 4 3.<
0
blsq ) {1 2 3}{2 2 3}.<
1
```

Note: Comparing values with different types may result in unexpected (but determinstic, thus not undefined) behaviour.

Sub .-

`Int a, Int b`: Integer subtraction.

```
blsq ) 1 5.-
-4
```

`Double a, Double b`: Double subtraction.

```
blsq ) 1.0 4.0.-
-3.0
```

`String a, String b`: Removes `b` from the end of `a` iff `b` is a suffix of `a`.

```
blsq ) "README.md" ".md".-
"README"
blsq ) "README.md" ".txt".-
"README.md"
```

`Int a, Block b`: Removes the first `a` elements from `b`.

```
blsq ) 3{1 2 3 4}.-
{4}
```

`String a, Int b`: Removes the first `b` characters from `a`.

```
blsq ) "abcd"2.-
"cd"
```

`Int a, String b`: Removes the first `a` characters from `b`.

```
blsq ) 2"abcd".-
"cd"
```

`Block a, Int b`: Removes the first `b` elements from `a`.

```
blsq ) {1 2 3 4}2.-
{3 4}
```

`Int a, Double b`: Converts `a` to Double, then subtracts.

```
blsq ) 4 3.0.-
1.0
```

`Double a, Int b`: Converts `b` to Double, then subtracts.

```
blsq ) 4.0 3.-
1.0
```

`Block a, Block b`: Removes `b` from the end of `a` iff `b` is a suffix of `a`.

```
blsq ) {1 2 3 4}{3 4}.-
{1 2}
blsq ) {1 2 3 4}{3 4 5}.-
{1 2 3 4}
```

Sum ++

`Block {}`: *Defined as: `0`*. The sum of an empty Block is zero.

`Block a`: *Defined as: `{.+}r[`*. Calculates the sum of a Block.

```
blsq ) {1 2 3 4}{.+}r[
10
blsq ) {1 2 3 4}++
10
blsq ) {1 2 3 4.0}++
10.0
```

`Int a, Int b`: Concatenates Integers (works on absolute values).

```
blsq ) 12 34++
1234
```

Swap j \/

Swaps the top two elements.

```
blsq ) 1 2
2
1
blsq ) 1 2j
1
2
```

Tail `[-`

`Block a:` Returns all but the first element of `a`.

```
blsq ) {1 2 3}[-
{2 3}
```

`String a:` Returns all but the first character of `a`.

```
blsq ) "hello"[-
"ello"
```

`Int a:` Returns all but the last digit of `a` (as Integer) (works on absolute value).

```
blsq ) 451[-
51
```

`Char a:` Convert to string.

```
blsq ) 'a[-
"a"
```

Unlines `un`

`Block {}:` If given an empty block returns an empty string.

```
blsq ) {}un
""
```

`Otherwise:` *Defined as:* `"\n"j[[\[`. This is the *inverse* of `ln` and inserts newlines between elements.

```
blsq ) {"abc" "def" "ghe"}"\n"j[[\[
"abc\ndef\nghe"
blsq ) {"abc" "def" "ghe"}un
"abc\ndef\nghe"
```

Authors' Notes: Due to its definition this built-in will only work as expected when the Block contains Strings. If you want to *unlines* a Block containing other types use `Su`.

```
blsq ) {1 2 3}un
"\n12\n3"
blsq ) {1 2 3}Su
"1\n2\n3"
```

UnlinesPretty `uN`

Defined as: ***`unsh`***.

```
blsq ) {"12" "23"}un
"12\n23"
blsq ) {"12" "23"}uN
12
23
blsq ) {"12" "23"}unsh
12
23
```

Unparse `up`

`Any a:` Converts a to display string. This is somewhat the *inverse* of `ps`.

```
blsq ) {1 2++}up
"{1 2 ++}"
```

Authors' Note: This built-in is somewhat equivalent to using `3SH`.

While `w!`

A while-loop

`Block f, Block p:` Executes `f` while `p` is not zero. `p` will be tested each time against the top of the stack.

```
blsq ) 5{+.}{10.<}w!
10
```

`Block f:` Executes `f` as long as the top of the stack is not zero. Same thing as doing `{code}{}w!`.

```
blsq ) 0 10{j+.+.j-.}w!
0
20
```

WithLines `wl`

Defined as: *`jlnjm[un`*. This built-in allows to map over the lines in a String.

```
blsq ) "abc\ndef"{<-}jlnjm[un
"cba\nfed"
blsq ) "abc\ndef"{<-}wl
"cba\nfed"
```

WithLinesParsePretty `wL`

Defined as: *`(ps)+]WL`*. This built-in allows to map over the lines in a String while calling *Parse* on each line automatically.

```
blsq ) "11 22\n5 6"{++Sh}(ps)+]WL
33
11
blsq ) "11 22\n5 6"{++Sh}wL
33
11
```

WithLinesPretty `WL`

Defined as: *`wlsh`*.

```
blsq ) "abc\ndef"{<-}wlsh
cba
fed
blsq ) "abc\ndef"{<-}WL
cba
fed
```

WithWords `ww`

Defined as: *`jWDjm[wd`*. This built-in allows to map over the words in a String.

```
blsq ) "hello world"{<-}jWDjm[wd
"olleh dlrow"
blsq ) "hello world"{<-}ww
"olleh dlrow"
```

WithWordsPretty `WW`

Defined as: *`wwsh`*.

```
blsq ) "hello world"{<-}wwsh
olleh dlrow
blsq ) "hello world"{<-}WW
olleh dlrow
```