Atoms and Definitions of Joy

March 25, 2003

Contents

Part I

Intro

March 1, 2003

Abstract

This document is containing a brief description
of all Atoms, defined by the JOY-System and
of all visible definitions, defined by the basic Joy Libraries.
The basic Libraries are
inilib, agglib, seqlib and numlib.
Furthermore it is containing a brief descriptions of all Atoms
defined in Typelib.joy and in Some.joy.
This document is up to date with Joy from March 1, 2003.

A	Aggregate: Set, String or List
B	Boolean: true or false
C	Character
F	Float
I	Integer
L	List
N	Number
P	Program
Seq	Sequence: String or List
Set	Set
Str	String
T	Tree
Sym	Symbol
X Y Z	Anything
->	Type specifier
=>	Evaluates to
...	Resulting type depends on parameter stack.

Table 1: Abbreviations

Joy Home Page [http://www.latrobe.edu.au/philosophy/phimvt/joy.html]

Generated at Tuesday 25-MAR-03 22:03:26 by Rabbit.

Rabbit is written in Joy and generates xhtml and latex output.*Rabbit* [http://groups.yahoo.com/group/concatenative/files/rabbit]

Part II

Briefer

Chapter 1

Briefer

1.1 Joy Types

list type	set type	string type
integer type	float type	
char type	truth value type	
file type		

1.2 Stack Manipulation

dup	dup2	dupd	
pop	pop2	popd	(pop3)
rolldown		rolldownd	
rollup		rollupd	
rotate		rotated	
swap	(swap2)	swapd	
(over)	(over2)	(overd)	
stack	newstack	unstack	
id	(nop)		

1.3 Aggregates

at	of	take
rest	restd	drop
size	setsize	
elements	set2string	string2set
unitlist	unitset	unitstring

cons	cons2	consd
uncons	uncons2	unconsd
swons	swons2	swonsd
unswons	unswons2	unswonsd
concat	swoncat	(concat3)
enconcat		
(wrapconcat)		(concatall)

first	firstd
second	secondd...
third	thirdd
fourth	
fifth	

(pair)	unpair	
pairlist	pairset	pairstring

has	in	all	some
null	null2	nulld	
small	equal	compare	(isin)
leaf	list	set	string

flatten	transpose	
qsort	qsort1	qsort1-1
mk_qsort		
reverse	reverselist	reversestring

delete		
insert	insertlist	insert-old
merge	merge1	
zip	(unzip)	(zipwith)

product	sum	scalarproduct
average	variance	cartproduct

from-to		
from-to-list	from-to-set	from-to-string
frontlist	frontlist1	
orlist	orlistfilter	
permlist	powerlist1	powerlist2
restlist	subseqlist	

treeshunt	treestrip
treeflatten	treereverse
treesample	

1.4 Numerics

sign	abs	pred	succ
even	odd		
null	null2	nulld	
small			
neg	negative	positive	
integer	numerical		

exp	e
log	log10

+	-	*	/
div	rem		
max	min		
	null2		

maxint	rand	srand

ceil	floor	trunc	modf
frexp	ldexp		
float	pi	power	sqrt
radians	celsius	fahrenheit	

fact	gcd	fib	prime
cube-root	qroots	deriv	
newton	use-newton		

sin	cos	tan
sindeg	cosdeg	tandeg
sinh	cosh	tanh
asin	acos	atan
		atan2

1.5 Logic

	=	!=	
<	<=	>	>=
not	and	or	xor
true	false	truth	falsity
	choice	negate	
conjoin	disjoin	sequand	sequor
	boolean	logical	

1.6 Characters

chr	ord	to-upper	to-lower
char			

1.7 Combinators

dip	dip2	dipd	dip3
(dipdd)	(dudip)	(sdip)	

branch	ifte
cond	
case	opcase

ifinteger	iffloat	
iflist	ifset	ifstring
ifchar	iflogical	iffile

while	repeat
times	
forever	

filter			
fold	fold2	foldr	foldr2
map		mapr	mapr2
spilt			
step	step2		stepr2
pairstep			
shunt			
interleave2	interleave2list		
treefilter	treemap	treestep	

primrec	tailrec	linrec	binrec
genrec	condlinrec		
treerec	treegenrec		

i	i2	x
all	some	
call		
cleave	infra	construct

app1			
app11	app12		
nullary	nullary2		
unary	unary2	unary3	unary4
binary			
ternary			

(b)	(sdip)
	(twice)
(dipdd)	(dudip)
(intersect)	(onitem)
(apps)	(foldapps)
(fold-andconds)	(fold-orconds)
(fold-listconcat)	(fold-stringconcat)

1.8 Time and Date

clock		
time	gmtime	localtime
today	now	strftime
mktime		
localtime-strings	show-todaynow	
month	weekdays	

1.9 Format

format	formatf
strtol	strtof

1.10 Stdin and Stdout

put	putch	putln	(putline)
putchars	putstrings	putlist	
newline			
get	ask		
space	bell		
stdin	stdout	stderr	

1.11 Files and Streams

fopen	fclose	fflush	ferror
feof	ftell	fseek	
fgetch	fgets		
fput	fputch	fputchars	fputstring
fread	fwrite		
frename	fremove		
file			

1.12 Joy

manual		
help	helpdetail	_help
echo	setecho	
autoput	setautoput	
undeferror	setundeferror	
__html_manual	__latex_manual	__manual_list

include	libload
all-libload	basic-libload
	special-libload

quit	abort
name	body
intern	
user	
gc	

(cp)	(ncp)	(cpifte)	(tpifte)

__memoryindex	__memorymax
__symtabindex	__symtabmax
__settracegc	
__dump	

1.13 System

argc argv
system
getenv

Part III

Atoms and Definitions of Standard Joy

Chapter 2

Joy Types

2.1 Aggregates

list type	-> L
set type	-> Set

string type	-> Str

list type :

-> L

The type of lists of values of any type (including lists) or the type of quoted programs which may contain operators or combinators. Literals of this type are written inside square brackets.

interp.c

```
[]                    =>   []
[ 3 512 -7]           =>   [ 3 512 -7]
[ john mary]          =>   [ john mary]
[ 'A 'C[ 'B]]         =>   [ 'A 'C[ 'B]]
[ dup *]              =>   [ dup *]
```

set type :

-> Set

The type of sets of small non-negative integers. The maximum is platform dependent, typically the range is 0..31. Literals are written inside curly braces.

interp.c

```
{}                    =>  {}
{ 0}                  =>  { 0}
{ 1 3 5}              =>  { 1 3 5}
{ 17 18 19}           =>  { 17 18 19}
```

string type :

`-> Str`

The type of strings of characters. Literals are written inside double quotes.

Unix style escapes are accepted: n - newline, t - tabulator and so on.

interp.c

```
""                  =>   ""
"A"                 =>   "A"
"hello world"       =>   "hello world"
"123"               =>   "123"
```

2.2 Numerics

float type `-> F`	integer type `-> I`

float type :

`-> F`

The type of floating-point numbers. Literals of this type are written with embedded decimal points (like 1.2) and optional exponent specifiers (like 1.5E2).

interp.c

```
1.20                =>   1.20
```

integer type :

`-> I`

The type of negative, zero or positive integers. Literals are written in decimal notation.

interp.c

```
-123                  =>   -123
0                     =>   0
42                    =>   42
```

2.3 Char, Truth, File

character type	`-> C`
file type	`-> STREAM`

truth value type	`-> B`

character type :

`-> C`

The type of characters. Literals are written with a single quote. Examples: `'A '7 ';` and so on. Unix style escapes are allowed.

interp.c

file type :

`-> STREAM`

The type of references to open I/O streams, typically but not necessarily files. The only literals of this type are stdin stdout and stderr.

interp.c

```
stdin                 =>   file type
stdout                =>   file type
stderr                =>   file type
```

truth value type :

`-> B`

The logical type or the type of truth values. It has just two literals: true and false.

interp.c

```
true                  =>   true
false                 =>   false
```

Chapter 3

Stack Manipulation

dup	`X -> X X`
dup2	`X Y -> X Y X Y`
dupd	`Y Z -> Y Y Z`
id	`->`
newstack	`... ->`
pop	`X ->`
pop2	`Y Z ->`
popd	`Y Z -> Z`
rolldown	`X Y Z -> Y Z X`

rolldownd	`X Y Z W -> Y Z X W`
rollup	`X Y Z -> Z X Y`
rollupd	`X Y Z W -> Z X Y W`
rotate	`X Y Z -> Z Y X`
rotated	`X Y Z W -> Z Y X W`
stack	`.. Y Z -> .. Y Z [Z Y ..]`
swap	`X Y -> Y X`
swapd	`X Y Z -> Y X Z`
unstack	`[X Y ..] -> ..Y X`

dup :

```
X -> X X
```

Pushes an extra copy of X onto stack.

interp.c

```
42 dup              =>   42 42
```

dup2 :

```
X Y -> X Y X Y
== dupd dup swapd;
```

inilib.joy

dupd :

```
Y Z -> Y Y Z
```

As if defined by: dupd == [dup] dip;

interp.c

id :

`->`

Identity function, does nothing. Any program of the form P id Q is equivalent to just P Q.

interp.c

newstack :

`... ->`

`== [] unstack;`

Remove the stack and continue with the empty stack.

inilib.joy

pop :

`X ->`

Removes X from top of the stack.

interp.c

```
 1 2 pop              =>   1
```

pop2 :

`Y Z ->`

`== pop pop ;`

inilib.joy

popd :

`Y Z -> Z`

As if defined by: popd == [pop] dip;

interp.c

```
 1 2 popd             =>   2
```

rolldown :

```
X Y Z -> Y Z X
```

Moves Y and Z down and moves X up.

interp.c

```
 1 2 3 rolldown      =>   2 3 1
```

rolldownd :

```
X Y Z W -> Y Z X W
```

As if defined by: rolldownd == [rolldown] dip;

interp.c

rollup :

```
X Y Z -> Z X Y
```

Moves X and Y up and moves Z down.

interp.c

```
 1 2 3 rollup        =>   3 1 2
```

rollupd :

```
X Y Z W -> Z X Y W
```

As if defined by: rollupd == [rollup] dip;

interp.c

rotate :

```
X Y Z -> Z Y X
```

Interchanges X and Z.

interp.c

```
 1 2 3 rotate        =>   3 2 1
```

rotated :

```
X Y Z W -> Z Y X W
```

As if defined by: rotated == [rotate] dip;

interp.c

stack :

```
.. Y Z -> .. Y Z [Z Y ..]
```

Pushes the stack as a list.

interp.c

```
 1 2 3 stack          =>   1 2 3[ 3 2 1]
```

swap :

```
 X Y -> Y X
```

Interchanges X and Y.

interp.c

```
 1 2 swap             =>   2 1
```

swapd :

```
X Y Z -> Y X Z
```

As if defined by: swapd == [swap] dip;

interp.c

unstack :

```
[X Y ..]  -> ..Y X
```

The list [X Y ..] becomes the new stack.

interp.c

```
 1 2 3[ 'a 'b]        =>   'b 'a
unstack
```

Chapter 4

Aggregates

4.1 at, of, drop, pair, rest, size, take, ...

at	`A I -> X`
drop	`A N -> A`
elements	`L -> Set`
of	`I A -> X`
rest	`A -> A`
restd	
set2string	

setsize	`-> setsize`
size	`A -> I`
string2set	
take	`A I -> A`
unitlist	`X -> L`
unitset	`I -> Set`
unitstring	`C -> Str`

at :

`A I -> X`

X is the member of A at position I. The first item is at position 0.

interp.c

```
[ 1 2 3] 0 at        =>   1
 0[ 1 2 3] of        =>   1
```

drop :

`A N -> A`

Result is A with its first N elements deleted.

interp.c

```
[ 1 2 3] 2 drop      =>  [ 3]
{ 1 2 3} 2 drop      =>  { 3}
 "abc" 2 drop        =>   "c"
[ 1 2 3] rest        =>  [ 2 3]
[ 1 2 3] 2 take      =>  [ 1 2]
```

elements :

`L -> Set`

`== {} swap [ swons] step ;`

Returns all members of L, doubles removed. The elements of L must fit the sets range, that is integers from 0 to 31.

agglib.joy

```
[ 1 2 3 1 2 3]       =>  { 1 2 3}
elements
```

of :

`I A -> X`

X is the member of A at position I. The first item is at position 0.

interp.c

```
[ 1 2 3] 0 at        =>   1
 0[ 1 2 3] of        =>   1
```

rest :

`A -> A`

Result is the non-empty aggregate A with its first member removed.

interp.c


```
[ 1 2 3] 2 drop      =>  [ 3]
{ 1 2 3} 2 drop      =>  { 3}
 "abc" 2 drop        =>   "c"
[ 1 2 3] rest        =>  [ 2 3]
[ 1 2 3] 2 take      =>  [ 1 2]
```


restd :

`== [rest] dip;`

agglib.joy

set2string :

```
== "" [[ chr] dip cons] foldr ;
```

agglib.joy

setsize :

```
-> setsize
```

Pushes the maximum number of elements in a set (platform dependent). Typically it is 32 and set members are in the range 0..31.

interp.c

```
 setsize            =>   32
```

size :

```
A -> I
```

Integer I is the size of aggregate A.

interp.c

string2set :

```
== {} swap shunt ;
```

agglib.joy

take :

```
A I -> A
```

Retain just the first I elements of A.

interp.c

```
[ 1 2 3] 2 drop      =>   [ 3]
{ 1 2 3} 2 drop      =>   { 3}
 "abc" 2 drop        =>    "c"
[ 1 2 3] rest        =>   [ 2 3]
[ 1 2 3] 2 take      =>   [ 1 2]
```

unitlist :

```
X -> L
== [] cons;
```

agglib.joy

```
 1 unitlist          =>  [ 1]
```

unitset :

`I -> Set`

`== {} cons;`

agglib.joy

unitstring :

`C -> Str`

`== '' cons;`

agglib.joy

4.2 cons and concat and related

concat	`A A -> A`
cons	`X A -> A`
cons2	`X Y A A -> A A`
consd	`X A Y -> A Y`
enconcat	`X A A -> A`
swoncat	`A A -> A`
swons	`A X -> A`
swons2	`A A X Y -> A A`

swonsd	`A X Y -> A Y`
uncons	`A -> X A`
uncons2	`A A -> X Y A A`
unconsd	`A X -> Y A X`
unswons	`A -> A X`
unswons2	`A A -> A A X Y`
unswonsd	`A X -> A Y X`

concat :

`A A -> A`

Evaluates to the concatenation of two aggregates.

interp.c

```
[ 1 2 3 4][ 3 4 5 6] concat => [ 1 2 3 4 3 4 5 6]
 "abcd" "efgh" concat => "abcdefgh"
{ 1 2 3 4}{ 3 4 5 6} concat => { 1 2 3 4 5 6}
```

cons :

`X A -> A`

Result is A with a new member X (first member for sequences).

interp.c

```
9[ 1 2 3] cons      =>  [ 9 1 2 3]
'z "abc" cons       =>   "zabc"
9{ 1 2 3} cons      =>  { 1 2 3 9}
```

cons2 :

`X Y A A -> A A`

`== swapd cons consd;`

Cons 2 values to 2 aggregates.

agglib.joy

```
1 2[] [] cons2      =>  [ 1] [ 2]
```

consd :

`X A Y -> A Y`

`== [ cons] dip ;`

agglib.joy

```
1[] 2 consd         =>  [ 1] 2
```

enconcat :

`X A A -> A`

The concatenation of two aggregates with X inserted between them.

As if defined by `enconcat == swapd cons concat;`

interp.c

```
0[ 1 2 3 4][ 3 4 5 6] enconcat => [ 1 2 3 4 0 3 4 5 6]
'0 "abcd" "efgh" enconcat => "abcd0efgh"
0{ 1 2 3 4}{ 3 4 5 6} enconcat => { 0 1 2 3 4 5 6}
```

swoncat :

`A A -> A`

`== swap concat;`

inilib.joy

```
[ 1 2 3 4][ 3 4 5 6] swoncat => [ 3 4 5 6 1 2 3 4]
 "abcd" "efgh" swoncat => "efghabcd"
{ 1 2 3 4}{ 3 4 5 6} swoncat => { 1 2 3 4 5 6}
```

swons :

`A X -> A`

Result is A with a new member X (first member for sequences).

interp.c

```
[ 1 2 3] 9 swons     =>  [ 9 1 2 3]
 "abc" 'z swons      =>   "zabc"
{ 1 2 3} 9 swons     =>  { 1 2 3 9}
```

swons2 :

`A A X Y -> A A`

`== swapd swons swonsd;`

Swons 2 items to 2 aggregates.

agglib.joy

```
 "ext"[ 1 2 3] 't    =>   "text"[ 999 1 2 3]
999 swons2
```

swonsd :

`A X Y -> A Y`

`== [ swons] dip ;`

agglib.joy

```
[] 1 2 swonsd        =>  [ 1] 2
```

uncons :

`A -> X A`

Returns the first and the rest of non-empty aggregate A.

interp.c

```
[ 1 2 3] uncons      =>   1[ 2 3]
```

uncons2 :

`A A -> X Y A A`

`== unconsd uncons swapd ;`

Uncons 2 values from 2 aggregates.

agglib.joy

```
[ 999 1 2 3]         =>   999 't[ 1 2 3] "ext"
"text" uncons2
```

unconsd :

`A X -> Y A X`

`== [ uncons] dip ;`

agglib.joy

```
[ 1 2 3] 99          =>   1[ 2 3] 99
unconsd
```

unswons :

`A -> A X`

Returns the rest and the first of non-empty aggregate A.

interp.c

```
[ 1 2 3] unswons     =>  [ 2 3] 1
```

unswons2 :

`A A -> A A X Y`

`== [ unswons] dip unswons swapd ;`

Unswons 2 items from 2 aggregates.

agglib.joy

```
[ 1 2 3] "text"      =>  [ 2 3] "ext" 1 't
unswons2
```

unswonsd :

`A X -> A Y X`

`== [ unswons] dip ;`

agglib.joy

```
[ 1 2 3] 99          =>  [ 2 3] 1 99
unswonsd
```

4.3 first, second,...

fifth	A -> X
first	A -> X
firstd	A X -> Y X
fourth	A -> X

second	A -> X
secondd	
third	A -> X
thirdd	A X -> Y X

fifth :

`A -> X`

`== 4 drop first ;`

agglib.joy

first :

`A -> X`

X is the first member of the non-empty aggregate A.

interp.c

firstd :

`A X -> Y X`

`== [ first] dip ;`

agglib.joy

fourth :

`A -> X`

`== 3 drop first ;`

agglib.joy

second :

`A -> X`

`== rest first ;`

agglib.joy

secondd :

`== [ secondd] dip ;`

Please use carefully!

agglib.joy

third :

`A -> X`

`== rest rest first ;`

agglib.joy

thirdd :

`A X -> Y X`

`== [ third] dip ;`

agglib.joy

4.4 pair and related

pairlist	`X X -> L`
pairset	`I I -> Set`

pairstring	`C C -> Str`
unpair	`A -> X Y`

pairlist :

`X X -> L`

`== [] cons cons;`

agglib.joy

pairset :

`I I -> Set`

`== {} cons cons;`

agglib.joy

pairstring :

`C C -> Str`

`== "" cons cons ;`

agglib.joy

unpair :

`A -> X Y`

`== uncons uncons pop;`

agglib.joy

`[ 1 2 3 4] unpair   =>   1 2`

4.5 Tests

all	`A [P:test] -> X`
compare	`A A -> I`
equal	`T T -> B`
has	`A X -> B`
in	`X A -> B`
leaf	`X -> B`
list	`X -> B`

null	`X -> B`
null2	`X Y -> B`
nulld	`X Y -> B Y`
set	`X -> B`
small	`X -> B`
some	`A [P:test] -> B`
string	`X -> B`

all :

`A [P:test] -> X`

Applies test P to members of aggregate A, returns true if all pass.

interp.c

```
[ 1 2 3 4][ 5 >]    =>   false
all
[ 1 2 3 4][ 5 <]    =>   true
all
```

compare :

`A A -> I`

I (=-1 0 +1) is the comparison of aggregates A1 and A2. The values correspond to the predicates <= = >=.

interp.c

```
{ 1 2 3}{ 1 2 3} compare => 0
{ 1 2 3}{ 1 2 4} compare => 4
{ 1 2 3}{ 1 31} compare => -3
```

```
"1 2 3" " 1 2 3" compare => 17
1 1 compare => 0
1 10 compare => -9
true false compare => 1
false true compare => -1
```

equal :

`T T -> B`

(Recursively) tests whether two trees are identical.

interp.c

```
[ 1[ 'a[ 5] 2]][ 1[ 'a[ 5] 2]] equal => true
[ 1[ 'a[ 5] 2]][ 1[ 'a[ 6] 2]] equal => false
```

has :

`A X -> B`

Tests whether aggregate A has X as a member.

interp.c

```
[ 1 2 3] 3 has      =>   true
 3[ 1 2 3] in       =>   true
```

in :

`X A -> B`

Tests whether X is a member of aggregate A.

interp.c

```
[ 1 2 3] 3 has      =>   true
 3[ 1 2 3] in       =>   true
```

leaf :

`X -> B`

Tests whether X is not a list.

interp.c

```
 1 leaf             =>   true
[] leaf             =>   false
```

list :

```
X -> B
```

Tests whether X is a list.

interp.c

null :

```
X -> B
```

Tests for empty aggregate X or zero numeric.

interp.c

```
 0 null             =>   true
[] null             =>   true
{} null             =>   true
 "" null            =>   true
```

null2 :

```
X Y -> B
```

```
== nulld null or;
```

Tests whether X or Y is null.

agglib.joy

nulld :

```
X Y -> B Y
```

```
== [ null] dip ;
```

agglib.joy

set :

```
X -> B
```

Tests whether X is a set.

interp.c

small :

```
X -> B
```

X has to be an aggregate or an integer.

Tests whether aggregate X has 0 or 1 members or integer X is 0 or 1.

interp.c

```
 -1 small          =>   true
 0 small           =>   true
 1 small           =>   true
 2 small           =>   false
[] small           =>   true
[ 1] small         =>   true
[ 1 2] small       =>   false
```

some :

`A [P:test] -> B`

Applies test to members of aggregate A and returns true if some (that is one or more) pass, false if not.

interp.c

```
[ 1 2 3][ odd]     =>   true
some
[ 2 4 6][ odd]     =>   false
some
```

string :

`X -> B`

Tests whether X is a string.

interp.c

4.6 L – L: flatten, qsort, reverse, transpose

flatten	`L -> L`
mk_qsort	`L [P] -> L`
qsort	`A -> A`
qsort1	`L -> L`
qsort1-1	`L -> L`

reverse	`S -> S`
reverselist	`L -> L`
reversestring	`Str -> Str`
transpose	`L -> L`

flatten :

`L -> L`

`== [ null] [] [ uncons] [ concat] linrec ;`

seqlib.joy

```
[[ 1 2 3][ 4 5        =>   [ 1 2 3 4 5 6 7[ 'a 'b] 9]
6][ 7[ 'a 'b] 9]]
flatten
```

mk_qsort :

`L [P] -> L`

Sort a sequence. The new order is obtained after applying P on every list item.

seqlib.joy

```
[[ 1 2 3][ 700        =>   [[ 700 -699][ 5][ 1 2 3]]
-699][ 5]][ sum]
mk_qsort
```

qsort :

`A -> A`

`== [small] [] [uncons [{>}] split] [swapd cons concat] binrec;`

Sort a sequence.

seqlib.joy

```
 "string." qsort      =>    ".ginrst"
[ 1 3 2] qsort        =>   [ 1 2 3]
```

qsort1 :

`L -> L`

`== [ small] [] [ uncons[[ first] unary2 >] split] [ swapd cons concat] binrec ;`

seqlib.joy

qsort1-1 :

`L -> L`

`== [ small] [] [ uncons unswonsd[ first >] split[ swons] dip2] [ swapd cons concat] binrec ;`

seqlib.joy

reverse :

`S -> S`

`== [[]] [''] iflist swap shunt;`

Reverse a list or string.

seqlib.joy

```
[ 1 2 3] reverse     =>   [ 3 2 1]
```

reverselist :

```
L -> L
== [] swap shunt ;
```

Reverse a list.

seqlib.joy

reversestring :

```
Str -> Str
== '' swap shunt ;
```

Reverse a string.

seqlib.joy

transpose :

```
L -> L
```

Transpose a list of lists.

seqlib.joy

```
[[ 1 2 3 4][ 'a 'b   =>   [[ 1 'a][ 2 'b][ 3 'c]]
'c]] transpose
```

4.7 L .. – N: average, product, sum, variance

average	`L|Set -> N`
cartproduct	`L L -> L`
product	`L|Set -> N`

scalarproduct	`L L -> N`
sum	`L|Set -> N`
variance	`L -> N`

average :

```
L|Set -> N
== [sum] [size] cleave /;
```

agglib.joy

cartproduct :

```
L L -> L
== [[]] dip2 [ pair swap[ swons] dip] pairstep ;
```

seqlib.joy

```
[ 1 2][ 'a 'b]       =>   [[ 2 'b][ 2 'a][ 1 'b][ 1 'a]]
cartproduct
```

product :

```
L|Set -> N
== 1 [ *] fold ;
```

Calculate the product of all list or set items.

seqlib.joy

```
[ 1 2 3 4 5]         =>   120
product
```

scalarproduct :

```
L L -> N
== [ 0] dip2 [ null2] [ pop2] [ uncons2[ * +] dip2] tailrec
;
```

Scalarproduct.

seqlib.joy

```
[ 2 3 1][ 6 4 12]    =>   36
scalarproduct
```

sum :

```
L|Set -> N
== 0[+]fold;
```

Calculate the sum of all list or set items.

agglib.joy

```
[ 1 2 3 4 5] sum     =>   15
```

variance :

`L -> N`

Calculate variance of lists items.

agglib.joy

4.8 L X – L: insert, delete, merge, zip

delete	`A X -> A`
insert	`A X -> A`
insert-old	
insertlist	`L X -> L`

merge	`Seq Seq -> Seq`
merge1	`Seq Seq -> Seq`
zip	`A A -> A`

delete :

`A X -> A`

```
== [[[ pop null][ pop]][[ firstd >][ pop]][[ firstd =][ pop
rest]][[ unconsd][ cons]]] condlinrec ;
```

Delete first occurrence of X out of A.

seqlib.joy

```
 "delete" 'e        =>   "dlete"
delete
```

insert :

`A X -> A`

```
== [ pop null] [ firstd >=] disjoin [ swons] [ unconsd] [ cons]
linrec ;
```

Insert X in A at sorted position.

seqlib.joy

```
[ 1 2 3 4 1] 3.50    =>  [ 1 2 3 3.50 4 1]
insert
```

insert-old :

Alternative implementation of insert.

seqlib.joy

insertlist :

```
L X -> L
```

seqlib.joy

`[ 1 2 3] "X" insertlist`	=>	`[[ "X" 1 2 3][ 1 "X" 2 3][ 1 2 "X" 3][ 1 2 3 "X"]]`

merge :

```
Seq Seq -> Seq
```

Concat 2 sorted sequences with sorted result.

seqlib.joy

`[ 1 2 3 4][ 2 3 4 5] merge`	=>	`[ 1 2 2 3 3 4 4 5]`
`"aabc" "abde" merge`	=>	`"aaabbcde"`

merge1 :

```
Seq Seq -> Seq
```

Concat 2 sorted sequences of sequences with sorted result.

seqlib.joy

zip :

```
A A -> A
```

```
== [null2] [pop2 []] [uncons2] [[pairlist] dip cons] linrec;
```

Zip 2 aggregates into a list of pairs.

agglib.joy

`[ 1 2 3][ 10 20 30] zip`	=>	`[[ 1 10][ 2 20][ 3 30]]`

4.9 Others

from-to	`I1|C1 I2|C2 A -> A`
from-to-list	`I I -> L`
from-to-set	`I I -> Set`
from-to-string	`C C -> Str`
frontlist	`A -> A`
frontlist1	`A -> A`
orlist	`[P] -> [P]`

orlistfilter	`[P] -> [P]`
permlist	`L -> L`
powerlist1	`L ->L`
powerlist2	
restlist	`L -> L`
subseqlist	`L -> L`

from-to :

`I1|C1 I2|C2 A -> A`

Create an aggregate containing values from I1 to I2 for sets and lists, from C1 to C2 for strings.

agglib.joy

```
3 7[] from-to       =>  [ 3 4 5 6 7]
3 7{} from-to       =>  { 3 4 5 6 7}
'c 'g "" from-to    =>   "cdefg"
```

from-to-list :

`I I -> L`

`== [] from-to;`

agglib.joy

```
5 10 from-to-list   =>  [ 5 6 7 8 9 10]
```

from-to-set :

`I I -> Set`

`== {} from-to;`

agglib.joy

```
5 10 from-to-set    =>  { 5 6 7 8 9 10}
```

from-to-string :

`C C -> Str`

`== '' from-to;`

agglib.joy

```
 'a 'f              =>   "abcdef"
from-to-string
```

frontlist :

```
A -> A
```

seqlib.joy

```
[ 1 2 3] frontlist  =>  [[][ 1][ 1 2][ 1 2 3]]
```

frontlist1 :

```
A -> A
```

```
== [ null] [[] cons] [ uncons] [[ cons] map popd[] swons] linrec
;
```

Thompson p 247

seqlib.joy

```
[ 1 2 3]            =>  [[][ 1][ 1 2][ 1 2 3]]
frontlist1
```

orlist :

```
[P] -> [P]
```

```
== [list] swap disjoin;
```

Creates a quoted program that evaluates to true, if [P] returns true or X is a list.

seqlib.joy

```
[ set] orlist       =>  [[ list][ true][ set] ifte]
{ 1 2 3}[ set]      =>   6
orlist[ sum][]
ifte
```

orlistfilter :

```
[P] -> [P]
```

```
== orlist[filter]cons;
```

seqlib.joy

```
[ set]                  =>  [[[ list][ true][ set] ifte] filter]
orlistfilter
[ 1{ 1 2}[ 99 88]       =>  [{ 1 2}[ 99 88]]
"string"][ set]
orlistfilter i
```

permlist :

`L -> L`

Create a list of all permutations of L.

seqlib.joy

```
[ 1 2 3] permlist       =>  [[ 1 2 3][ 2 1 3][ 2 3 1][ 1 3 2][ 3 1 2][ 3 2
                            1]]
```

powerlist1 :

`L ->L`

seqlib.joy

```
[ 1 2 3]                =>  [[][ 3][ 2][ 2 3][ 1][ 1 3][ 1 2][ 1 2 3]]
powerlist1
```

powerlist2 :

seqlib.joy

```
[ 1 2 3]                =>  [[ 1 2 3][ 1 2][ 1 3][ 1][ 2 3][ 2][ 3][]]
powerlist2
```

restlist :

`L -> L`

`== [ null] [[] cons] [ dup rest] [ cons] linrec ;`

seqlib.joy

```
[ 1 2 3 4]              =>  [[ 1 2 3 4][ 2 3 4][ 3 4][ 4][]]
restlist
```

subseqlist :

`L -> L`

seqlib.joy

```
[ 1 2 3]            =>  [[ 1][ 1 2][ 1 2 3][ 2][ 2 3][ 3][]]
subseqlist
```

4.10 Trees

treeflatten treereverse treesample

treeshunt treestrip

treeflatten :

seqlib.joy

treereverse :

seqlib.joy

treesample :

```
== [[ 1 2[ 3 4] 5[[[ 6]]] 7] 8] ;
```

seqlib.joy

treeshunt :

seqlib.joy

treestrip :

seqlib.joy

Chapter 5

Numerics

5.1 Unary

abs	`N -> N`
even	`N -> B`
integer	`X -> B`
neg	`N -> N`
negative	`N -> B`
null	`X -> B`
nulld	`X Y -> B Y`

numerical	`X -> B`
odd	`N -> B`
positive	`N -> B`
pred	`I -> I`
sign	`I -> I`
small	`X -> B`
succ	`I -> I`

abs :

`N -> N`

Result is the absolute value (0 1 2..) of number N. Also supports float.

interp.c

even :

`N -> B`

`== 2 rem null;`

Tests whether number N is even.

numlib.joy

```
2 even          =>   true
3 even          =>   false
```

integer :

`X -> B`

Tests whether X is an integer.

interp.c

neg :

`N -> N`

Result is the negative of a number. Also supports float.

interp.c

negative :

`N -> B`

`== 0 < ;`

Returns true if N is negative.

numlib.joy

null :

`X -> B`

Tests for empty aggregate X or zero numeric.

interp.c

nulld :

`X Y -> B Y`

`== [ null] dip ;`

agglib.joy

numerical :

`X -> B`

True if X is an integer or a float.

inilib.joy

odd :

`N -> B`

`== even not;`

Tests whether a number is not even.

numlib.joy

```
 2 odd              =>   false
 3 odd              =>   true
```

positive :

`N -> B`

`== 0 > ;`

Returns true if N is greater that 0.

numlib.joy

```
 0 positive         =>   false
 1 positive         =>   true
```

pred :

`I -> I`

Result is the predecessor of an integer.

interp.c

```
 1 pred             =>   0
```

sign :

`I -> I`

Result is the sign (-1 or 0 or +1) of a number. Also supports float.

interp.c

```
 -7 sign            =>   -1
 0 sign             =>   0
 7.20 sign          =>   1.00
```

small :

`X -> B`

X has to be an aggregate or an integer.

Tests whether aggregate X has 0 or 1 members or numeric 0 or 1.

interp.c

```
 -1 small            =>   true
 0 small             =>   true
 1 small             =>   true
 2 small             =>   false
[] small             =>   true
[ 1] small           =>   true
[ 1 2] small         =>   false
```

succ :

`I -> I`

Returns the successor of an integer.

interp.c

```
 1 succ              =>   2
```

5.2 Binary

	`N N -> N`
+	`N N -> N`
-	`N N -> N`
/	`N N -> N`
div	`I I -> I I`

max	`N N -> N`
min	`N N -> N`
null2	`X Y -> B`
rem	`I I -> I`

***** :

`N N -> N`

Result is the product of two numbers. Also supports float.

interp.c

+ :

`N N -> N`

Result is the adding of two numbers. Also supports float.

interp.c

- :

`N N -> N`

Numeric N is the result of subtracting two numbers. Also supports float.

interp.c

/ :

```
N N -> N
```

Result is the (rounded) ratio of two integers or the quotient of two numbers. Also supports float.

interp.c

```
10 3 /              =>   3
10.00 3 /           =>   3.33
10 3.00 /           =>   3.33
10 3 div            =>   3 1
10 3 rem            =>   1
```

div :

```
I I -> I I
```

Result are the quotient and remainder of dividing two integers.

interp.c

```
7 3 rem             =>   1
7 3 div             =>   2 1
```

max :

```
N N -> N
```

Result is the maximum of two numbers. Also supports float.

interp.c

min :

```
N N -> N
```

Result is the minimum of two numbers. Also supports float.

interp.c

null2 :

```
X Y -> B
```

`== nulld null or;`

Tests whether X or Y is null.

agglib.joy

rem :

`I I -> I`

Result is the remainder of dividing two integers. Also supports float.

interp.c

```
7 3 div          =>   2 1
7 3 rem          =>   1
```

5.3 Float

ceil	`F -> F`
celsius	`F -> F`
fahrenheit	`F -> F`
float	`X -> B`
floor	`F -> F`
frexp	`F -> F I`
ldexp	`F I -> F`

modf	`F -> F F`
pi	`-> F`
pow	`F1 F2 -> F`
radians	`N -> F`
sqrt	`F -> F`
trunc	`F -> I`

ceil :

`F -> F`

Result is the float ceiling of F.

interp.c

```
1.25 ceil        =>   2.00
1.52 ceil        =>   2.00
1.25 floor       =>   1.00
1.52 floor       =>   1.00
1.25 trunc       =>   1
1.52 trunc       =>   1
```

celsius :

`F -> F`

`== 32 - 5 * 9 / ;`

Convert Fahrenheit to Celsius.

numlib.joy

fahrenheit :

`F -> F`

`== 9 * 5 / 32 + ;`

Convert Celsius to Fahrenheit.

numlib.joy

float :

`X -> B`

Tests whether X is a float.

interp.c

floor :

`F -> F`

Result is the floor of F.

interp.c

```
1.25 ceil          =>   2.00
1.52 ceil          =>   2.00
1.25 floor         =>   1.00
1.52 floor         =>   1.00
1.25 trunc         =>   1
1.52 trunc         =>   1
```

frexp :

`F -> F I`

Result is the mantissa and the exponent of F. Unless F = 0 0.5 <= abs(F2) < 1.0.

interp.c

ldexp :

`F I -> F`

Result is F times 2 to the Ith power.

interp.c

modf :

`F -> F F`

Results are the fractional part and the integer part (but expressed as a float) of F.

interp.c

```
12.25 modf          =>   0.25 12.00
```

pi :

`-> F`

`== 3.14159265;`

numlib.joy

pow :

`F1 F2 -> F`

Result is F1 raised to the F2th power.

interp.c

radians :

`N -> F`

`== pi * 180 /;`

Convert degree to radians.

numlib.joy

sqrt :

`F -> F`

Result is the square root of F.

interp.c

trunc :

`F -> I`

Result is an integer equal to the float F truncated toward zero.

interp.c

```
1.25 ceil           =>   2.00
1.52 ceil           =>   2.00
1.25 floor          =>   1.00
1.52 floor          =>   1.00
1.25 trunc          =>   1
1.52 trunc          =>   1
```

5.4 Trigonometric

acos	`F -> F`
asin	`F -> F`
atan	`F -> F`
atan2	`F F -> F`
cos	`F -> F`
cosdeg	`F -> F`
cosh	`F -> F`

sin	`F -> F`
sindeg	`F -> F`
sinh	`F -> F`
tan	`F -> F`
tandeg	`F -> F`
tanh	`F -> F`

acos :

`F -> F`

Result is the arc cosine of F.

interp.c

asin :

`F -> F`

Result is the arc sine of F.

interp.c

atan :

`F -> F`

Result is the arc tangent of F.

interp.c

atan2 :

`F F -> F`

Result is the arc tangent of the quotient of two floats.

interp.c

cos :

`F -> F`

Result is the cosine of F.

interp.c

cosdeg :

`F -> F`

`== radians cos;`

Cosine calculated from degree-value.

numlib.joy

cosh :

`F -> F`

Result is the hyperbolic cosine of F.

interp.c

sin :

`F -> F`

Result is the sine of F.

interp.c

sindeg :

`F -> F`

`== radians sin;`

Sine calculated from degree-value.

numlib.joy

sinh :

`F -> F`

Result is the hyperbolic sine of F.

interp.c

tan :

`F -> F`

Result is the tangent of F.

interp.c

tandeg :

`F -> F`

`== radians tan;`

Tangent calculated from degree-value.

numlib.joy

tanh :

`F -> F`

Result is the hyperbolic tangent of F.

interp.c

5.5 Logarithm

e	`-> F`
exp	`F -> F`

log	`F -> F`
log10	`F -> F`

e :

`-> F`

`== 1.0 exp;`

numlib.joy

exp :

`F -> F`

Result is e (2.718281828...) raised to the Fth power.

interp.c

log :

`F -> F`

Result is the natural logarithm of F.

interp.c

log10 :

`F -> F`

F is the common logarithm of F.

interp.c

5.6 Random, Maxint

maxint	`-> maxint`
rand	`-> I`

srand	`I ->`

maxint :

`-> maxint`

Pushes largest integer (platform dependent). Typically it is 32 bits.

interp.c

```
2147483647          =>    2147483647
```

rand :

`-> I`

I is a random integer.

interp.c

srand :

`I ->`

Sets the random integer seed to integer I.

interp.c

5.7 Algorithm: fact, fib, ...

cube-root	`F -> F`
deriv	
fact	
fib	
gcd	

newton	
prime	
qroots	`F:a F:b F:c ->`
use-newton	

cube-root :

`F -> F`

Calculate cube root using newton

numlib.joy

```
64 cube-root         =>   4.00
```

deriv :

numlib.joy

fact :

numlib.joy

```
5 fact               =>   120
```

fib :

numlib.joy

gcd :

numlib.joy

```
8 12 gcd             =>   4
```

newton :

numlib.joy

prime :

numlib.joy

```
12 prime             =>   false
17 prime             =>   true
```

qroots :

```
F:a F:b F:c ->
```

Find roots of the quadratic equation with coefficients a b c :

a * x*x + b * x + c = 0

numlib.joy

```
 1 2 -3 qroots       =>  [ 1.00 -3.00]
 1 0 0 qroots        =>  [ 0.00]
 1 2 3 qroots        =>  [ _COMPLEX]
```

use-newton :

numlib.joy

Chapter 6

Logic

!=	`X Y -> B`	false	`-> B`
<	`X Y -> B`	falsity	`-> B`
<=	`X Y -> B`	logical	`X -> B`
=	`X Y -> B`	negate	`X -> X [P]`
>	`X Y -> B`	not	`X -> Y`
>=	`X Y -> B`	or	`X Y -> Z`
and	`X Y -> Z`	sequand	`X [P:cond] [P:true] -> ...`
boolean	`X -> B`	sequor	`X [P:cond] [P:false] -> ...`
choice	`B X Y -> Z`	true	`-> B`
conjoin	`[P] [P] -> [P]`	truth	`-> B`
disjoin	`[P] [P] -> [P]`	xor	`X Y -> Z`

!= :

`X Y -> B`

Either both X and Y are numeric or both are strings or symbols. Tests whether X not equal to Y. Also supports float.

interp.c

< :

`X Y -> B`

Either both X and Y are numeric or both are strings or symbols. Tests whether X less than Y. Also supports float.

interp.c

<= :

`X Y -> B`

Either both X and Y are numeric or both are strings or symbols. Tests whether X less than or equal to Y. Also supports float.

interp.c

= :

`X Y -> B`

Either both X and Y are numeric or both are strings or symbols. Tests whether X equal to Y. Also supports float.

interp.c

> :

`X Y -> B`

Either both X and Y are numeric or both are strings or symbols. Tests whether X greater than Y. Also supports float.

Qsort is using `>` in order to obtain the arrangement.

interp.c

```
[ "dup" dup2         =>  [ "1" "Xx" "dup" dup dup2 "dup2" "dup3"]
"dup2" "dup3" dup
"1" "Xx"] qsort
```

>= :

`X Y -> B`

Either both X and Y are numeric or both are strings or symbols. Tests whether X greater than or equal to Y. Also supports float.

interp.c

and :

`X Y -> Z`

Result is the intersection of two sets or the logical conjunction of two truth values.

interp.c

```
{ 1 2 3}{ 2 3 4}     =>  { 2 3}
and
 true false and      =>   false
```

boolean :

`X -> B`

True if X is a logical or a set.

inilib.joy

choice :

`B X Y -> Z`

Result is X if B is true and Y if B is false.

interp.c

```
 true 1 2 choice     =>   1
 false 1 2 choice    =>   2
```

conjoin :

`[P][P] -> [P]`

`== [[false] ifte] cons cons;`

inilib.joy

```
[ P1][ P2] conjoin   =>  [[ P1][ P2][ false] ifte]
 17[ 10 >][ 20 <]    =>   true
conjoin i
```

disjoin :

`[P][P] -> [P]`

`== [ifte] cons [true] swons cons;`

inilib.joy

```
[ P1][ P2] disjoin   =>  [[ P1][ true][ P2] ifte]
 17[ 10 <][ 20 >]    =>   false
disjoin i
```

false :

`-> B`

Pushes the value false.

interp.c

```
 false               =>   false
```

falsity :

```
-> B
```

```
== false;
```

inilib.joy

logical :

```
X -> B
```

Tests whether X is a logical.

interp.c

negate :

```
X -> X [P]
```

```
== [[false] [true] ifte] cons;
```

inilib.joy

```
[ small] negate      =>  [[ small][ false][ true] ifte]
[][ small] negate    =>  [] false
i
```

not :

```
X -> Y
```

Y is the complement of a set, the logical negation of a truth value.

interp.c

```
{ 1 2 3} not         =>  { 0 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
                         20 21 22 23 24 25 26 27 28 29 30 31}
 true not            =>   false
```

or :

```
X Y -> Z
```

Z is the union of two sets, the logical disjunction of two truth values.

interp.c

```
{ 1 2 3}{ 2 3 4}     =>  { 1 2 3 4}
or
 true false or       =>   true
```

sequand :

```
X [P:cond][P:true] -> ...
```

```
== [pop false] ifte;
```

inilib.joy

sequor :

```
X [P:cond][P:false] -> ...
```

```
== [pop true] swap ifte;
```

inilib.joy

true :

```
-> B
```

Pushes the value true.

interp.c

```
 true                   =>   true
```

truth :

```
-> B
```

```
== true;
```

inilib.joy

xor :

```
X Y -> Z
```

Z is the symmetric difference of two sets, the logical exclusive disjunction of two truth values.

interp.c

```
{ 1 2 3}{ 2 3 4}      =>  { 1 4}
xor
 true false xor       =>   true
```

Chapter 7

Characters

char	`X -> B`
chr	`I -> C`
ord	`C -> I`

to-lower	`C -> C`
to-upper	`C -> C`

char :

`X -> B`

Tests whether X is a character.

interp.c

chr :

`I -> C`

C is the character whose ASCII value is integer I (or logical or character).

interp.c

```
97 chr              =>   'a
```

ord :

`C -> I`

Integer I is the ASCII value of character C (or logical or integer).

interp.c

```
'a ord              =>   97
```

to-lower :

```
C -> C
```

```
== [ 'a <] [ 32 +] [] ifte ;
```

inilib.joy

```
 'c to-lower         =>   'c
 'X to-lower         =>   'x
 '0 to-lower         =>   'P
```

to-upper :

```
C -> C
```

```
== [ 'a >=] [ 32 -] [] ifte ;
```

inilib.joy

```
 'c to-upper         =>   'C
 'X to-upper         =>   'X
```

Chapter 8

Combinators

8.1 Dips

dip	`X [P] -> ... X`	dip3	`X Y Z [P] -> ... X Y Z`
dip2	`X Y [P] -> ... X Y`	dipd	`X Y [P] -> ... X Y`

dip :

`X [P] -> ... X`

Saves X, executes P and pushes X back.

interp.c

```
 1 2 3[ "diver"]    =>   1 2 "diver" 3
dip
```

dip2 :

`X Y [P] -> ... X Y`

`== [dip] cons dip;`

Saves X and Y, executes P and restores X and Y.

Equivalent to dipd.

inilib.joy

```
 1 2 3[ "diver"]    =>   1 "diver" 2 3
dip2
```

dip3 :

```
X Y Z [P] -> ...  X Y Z
```

```
== [dip2]cons dip;
```

Saves X Y Z, executes P and restores X Y Z.

inilib.joy

```
 1 2 3[ "diver"]    =>   "diver" 1 2 3
dip3
```

dipd :

```
X Y [P] -> ...  X Y
```

```
== [dip] cons dip;
```

Saves X and Y, executes P and restores X and Y.

Equivalent to dip2.

inilib.joy

```
 1 2 3[ "diver"]    =>   1 "diver" 2 3
dipd
```

8.2 Branches

branch	`B [P1] [P2] -> ...`
case	`X [[Xi Pi]..  [P:default]] -> ...`
cond	`[[[cond] true]..  [Default]] -> ...`
conts	`-> [[P] [Q] ..]`
ifte	`[P:if] [P:true] [P:false] -> ...`
opcase	`X [..[X Xs]..]  -> [Xs]`

branch :

```
B [P1] [P2] -> ...
```

If B is true then executes P1 else executes P2.

interp.c

```
 true[ "true"][     =>   "true"
"false"] branch
 true[][ "true"][   =>   true "true"
"false"] ifte
```

case :

```
X [[Xi Pi].. [P:default]] -> ...
```

Indexing on the value of X, execute the matching Pi or P:default.

interp.c

```
1[[ 1 "one"][ 2 "two"][ "default"]] case => "one"
```

```
2[[ 1 "one"][ 2 "two"][ "default"]] case => "two"
```

```
'D[[ 1 "one"][ 2 "two"][ "default"]] case => 'D "default"
```

```
9[[ 1[ "one"]][ 2 "two"][ 9 "default"]] case => 9 9 "default"
```

```
1[[ 1 4 5 *][ 2 "two"][ "default"]] case => 20
```

```
1[[ 1[ "one"]][ 2 "two"][ "default"]] case => [ "one"]
```

cond :

```
[[[cond] true].. [Default]] -> ...
```

Tries each condition. If a condition yields true executes the corresponding if-true and exits.

If no condition yields true executes default.

interp.c

```
1[[[ 1 =] pop "One"][[ 2 =] pop "Two"][ pop "Default"]] cond
=> "One"
```

```
2[[[ 1 =] pop "One"][[ 2 =] pop "Two"][ pop "Default"]] cond
=> "Two"
```

```
'D[[[ 1 =] pop "One"][[ 2 =] pop "Two"][ pop "Default"]] cond
=> "Default"
```

```
1[[[ 1 =] pop 4 5 *][[ 2 =] pop "Two"][ pop "Default"]] cond
=> 20
```

```
1[[[ 1 =] pop[ "One"]][[ 2 =] pop "Two"][ pop "Default"]] cond
=> [ "One"]
```

conts :

```
-> [[P] [Q] ..]
```

Pushes current continuations. Buggy, do not use.

interp.c

ifte :

```
[P:if] [P:true] [P:false] -> ...
```

Executes P:condition. If that yields true then executes P:if-true else executes P:if-false.

interp.c

```
 1234[ 1234 =][      =>   1234 "true"
"true"][ "false"]
ifte
 1234[ 1234 =][      =>   "true"
pop "true"][ pop
"false"] ifte
 1234[ odd][][ 1     =>   1235
+] ifte
```

opcase :

```
X [..[X Xs]..]  -> [Xs]
```

Indexing on type of X returns the list [Xs].

interp.c

```
 1[[ 1 "integer"][ 'c "char"][[] "list"][ "default"]] opcase
=> 1[ "integer"]

 2[[ 1 "integer"][ 'c "char"][[] "list"][ "default"]] opcase
=> 2[ "integer"]

 'a[[ 1 "integer"][ 'c "char"][[] "list"][ "default"]] opcase
=> 'a[ "char"]

[ 1][[ 1 "integer"][ 'c "char"][[] "list"][ "default"]] opcase
=> [ 1][ "list"]

{}[[ 1 "integer"][ 'c "char"][[] "list"][ "default"]] opcase
=> {}[ "default"]
```

8.3 Branches on Type Tests

ifchar	`X [P1] [P2] -> ...`
iffile	`X [P1] [P2] -> ...`
iffloat	`X [P1] [P2] -> ...`
ifinteger	`X [P1] [P2] -> ...`
iflist	`X [P1] [P2] -> ...`
iflogical	`X [P1] [P2] -> ...`
ifset	`X [P1] [P2] -> ...`
ifstring	`X [P:if-true] [P:if-false] -> ...`

ifchar :

`X [P1] [P2] -> ...`

If X is a character executes P1 else executes P2.

interp.c

```
 'c[ true][ false]   =>   'c true
ifchar
 3[ true][ false]    =>   3 false
ifchar
```

iffile :

`X [P1] [P2] -> ...`

If X is a file executes P1 else executes P2.

interp.c

iffloat :

`X [P1] [P2] -> ...`

If X is a float executes P1 else executes P2.

interp.c

ifinteger :

`X [P1] [P2] -> ...`

If X is an integer executes P1 else executes P2.

interp.c

iflist :

`X [P1] [P2] -> ...`

If X is a list executes P1 else executes P2.

interp.c

iflogical :

`X [P1] [P2] -> ...`

If X is a logical or truth value executes P1 else executes P2.

interp.c

ifset :

`X [P1] [P2] -> ...`

If X is a set executes P1 else executes P2.

interp.c

ifstring :

`X [P:if-true] [P:if-false] -> ...`

If X is a string executes P:if-true else executes P:if-false.

interp.c

```
 "text"[ "true"][    =>   "text" "true"
"false"] ifstring
```

8.4 Loops

forever	`[P] -> ...`
repeat	`[P:body][P:condition] -> ...`
times	`N [P] -> ...`
while	`[P:condition] [P:body] -> ...`

forever :

`[P] -> ...`

`== 2147483647 swap times ;`

inilib.joy

repeat :

```
[P:body][P:condition] -> ...
```

```
== dupd swap [ i] dip2 while ;
```

Execute P:body. Then: While executing P:condition yields true executes P:body.

inilib.joy

```
 1[ succ][ 10 <]     =>   10
repeat
 1[ succ][ 10 =]     =>   2
repeat
```

times :

```
N [P] -> ...
```

Executes N times P.

interp.c

```
 4[ 'a] times        =>   'a 'a 'a 'a
```

while :

```
[P:condition] [P:body] -> ...
```

While executing P:condition yields true executes P:body.

interp.c

```
 1[ 10 <][ succ]     =>   10
while
 1[ 10 =][ succ]     =>   1
while
```

8.5 Functionals on Aggregates

filter	`A [P] -> A`
fold	`A X [P] -> X`
fold2	`A A X [P] -> A`
foldr	`A X [P] -> X`
foldr2	
interleave2	
interleave2list	
map	`A [P] -> A`
mapr	`A [P] -> A`

mapr2	
pairstep	
shunt	`A A -> A`
split	`A [P] -> A A`
step	`A [P] -> ...`
stepr2	`A A [P] -> ...`
treefilter	
treemap	
treestep	`T [P] -> ...`

filter :

`A [P] -> A`

Uses test P to filter an aggregate and produces a same-type aggregate.

interp.c

```
[ 1 2 3 4][ odd]     =>   [ 1 3]
filter
```

fold :

`A X [P] -> X`

Starting with X sequentially pushes members of aggregate A and combines with binary operator P to produce new X.

interp.c

```
[ 1 2 3 4] 0[ +]     =>    10
fold
[ 1 2 3 4] 0[]       =>    0 1 2 3 4
fold
 0[ 1 2 3 4][ +]     =>    10
step
```

fold2 :

`A A X [P] -> A`

`== rollupd stepr2 ;`

Starting with X sequentially pushes members of both aggregates and combines with ternary operator P to produce new X.

agglib.joy

```
[ 1 2 3][ 6 7        =>   [[ 3 8][ 2 7][ 1 6]]
8][][ pair swons]
fold2
```

foldr :

```
A X [P] -> X
== [[[ null]] dip[] cons[ pop] swoncat[ uncons]] dip linrec
;
```

agglib.joy

```
[ 100 3 10] 0[ -]    =>   107
foldr
[ 100 3 10] 0[ -]    =>   -113
fold
```

foldr2 :

```
== [[[ null2]] dip[] cons[ pop2] swoncat[ uncons2]] dip linrec
;
```

agglib.joy

interleave2 :

```
== [cons cons] foldr2;
```

agglib.joy

interleave2list :

```
== [] interleave2;
```

agglib.joy

map :

```
A [P] -> A
```

Executes P on each member of aggregate A and collects results in a same-type aggregate.

interp.c

```
[ 1 2 3 4][ odd]     =>   [ true false true false]
map
```

mapr :

```
A [P] -> A
```

agglib.joy

```
[ 1 2 3 4][ odd]     =>   [ true false true false]
mapr
```

mapr2 :

```
== [[ null2][ pop2[]][ uncons2]] dip [ dip cons] cons linrec
;
```

agglib.joy

pairstep :

```
== [ dupd] swoncat [ step pop] cons cons step ;
```

agglib.joy

shunt :

```
A A -> A
```

```
== [swons] step;
```

agglib.joy

```
[][ 1 2 3] shunt     =>   [ 3 2 1]
```

split :

```
A [P] -> A A
```

Uses test P to split aggregate A into two aggregates of the same type.

interp.c

```
[ 1 2 3 4][ odd]     =>   [ 1 3][ 2 4]
split
```

step :

```
A [P] -> ...
```

Sequentially putting members of aggregate A onto stack, executes P for each member of A.

interp.c

```
[ 1 2 3 4][ odd]     =>   true false true false
step
 0[ 1 2 3 4][ +]     =>   10
step
[ 1 2 3 4] 0[ +]     =>   10
fold
```

stepr2 :

```
A A [P] -> ...
```

```
== [[ null2][ pop pop]] dip [ dip] cons [ dip] cons [ uncons2]
swoncat tailrec ;
```

agglib.joy

```
[ 1 2 3][ 4 5 6      =>  [ 1 4][ 2 5][ 3 6]
0][ pair] stepr2
```

treefilter :

seqlib.joy

treemap :

seqlib.joy

treestep :

```
T [P] -> ...
```

Recursively traverses leaves of tree T executes P for each leaf.

interp.c

8.6 Recursive

binrec	`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`
condlinrec	`[ [C1] [C2] .. [P:default] ] -> ...`
genrec	`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`
linrec	`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`
primrec	`X [P:initial] [P:combine] -> X`
tailrec	`[P:condition] [P:if-true] [P:R1] -> ...`
treegenrec	`T [P:O1] [P:O2] [P:C] -> ...`
treerec	`T [P:O] [P:C] -> ...`

binrec :

`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`

Executes P:condition. If that yields true executes P:if-true.

Else uses P:R1 to produce two intermediates, recurses on both and then executes P:R2 to combines their results.

interp.c

```
[ 1 3 5 2 4][        =>  [ 1 2 3 4 5]
small][][ uncons[
>] split][
enconcat] binrec
```

condlinrec :

`[ [C1] [C2] .. [P:default] ] -> ...`

Each `[Ci]` is of the forms `[[P:condition] [P:if-true]]` or `[[P:condition] [P:R1] [P:R2]]`.

Tries each P:condition. If that yields true and there is just a P:if-true executes that and exits.

If there are P:R1 and P:R2 executes P:R1, recurses and executes P:R2.

Subsequent case are ignored.

If no P:condition yields true then [P:default] is used. It is of the forms [[T]] or [[R1] [R2]].

For the former executes T.

For the latter executes R1, recurses and executes R2.

interp.c

genrec :

`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`

Executes P:condition. If that yields true executes P:if-true.

Else executes P:R1 and then

`[[P:condition] [P:if-true] [P:R1] [P:R2] genrec] P:R2`.

interp.c

linrec :

`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`

Executes P:condition. If that yields true executes P:if-true. Else executes P:R1, recurses and executes P:R2.

interp.c

```
 1[ 1 2 3 4 5][      =>   120
null][ pop][
uncons][ *] linrec
```

primrec :

`X [P:initial] [P:combine] -> X`

Executes P:initial to obtain an initial value.

If X is an integer uses increasing positive integers from 1 up to X and combines by P:combine for new X.

For aggregate X uses successive members and combines by P:combine for new X.

interp.c

```
 5[ 1][ *] primrec   =>   120
[ 1 2 3 4 5][ 1][    =>   120
*] primrec
```

tailrec :

`[P:condition] [P:if-true] [P:R1] -> ...`

Executes P:condition. If that yields true executes P:if-true.

Else executes P:R1 and recurses.

interp.c

```
 1[ 1 2 3 4 5][      =>   120
null][ pop][
uncons[ *] dip]
tailrec
```

treegenrec :

`T [P:O1] [P:O2] [P:C] -> ...`

T is a tree. If T is a leaf executes P:O1.

Else executes P:O2 and then

`[[P:O1] [P:O2] [P:C] treegenrec] P:C`.

interp.c

treerec :

`T [P:O] [P:C] -> ...`

T is a tree. If T is a leaf executes P:O.

Else executes `[[O] [C] treerec] C`.

interp.c

8.7 Apply: i, i2, b, x, infra, cleave,...

all	`A [P:test] -> B`
call	`Symbol -> ...`
cleave	`X [P] [P] -> Y Z`
construct	`[P:init] [[P1] [P2] ..]  -> X1 X2 ..`
i	`[P] -> ...`
i2	`...  X [P1][P2] -> ...`
infra	`L [P] -> L`
some	`A [P:test] -> B`
x	`[P] -> ...`

all :

`A [P:test] -> B`

Applies test P to members of aggregate A, returns true if all pass.

interp.c

```
[ 1 2 3 4][ 5 >]      =>    false
all
[ 1 2 3 4][ 5 <]      =>    true
all
```

call :

`Symbol -> ...`

`== [] cons i ;`

Execute the symbol.

inilib.joy

```
 1 2 "+" intern       =>    3
call
 1 2[ +] first        =>    3
call
```

cleave :

`X [P] [P] -> Y Z`

Executes both quotations, each with X on top of the stack and each producing one result.

interp.c

```
 3[ 2 *][ 3 *]        =>    6 9
cleave
```

construct :

`[P:init] [[P1] [P2] ..] -> X1 X2 ..`

Saves state of stack and then executes [P:init]. Then executes each [Pi] to give Xi pushed onto saved stack.

interp.c

```
 4 5[ pop][[ 10       =>    4 5 14 24 30
+][ 20 +][ 30]]
construct
```

i :

`[P] -> ...`

Executes P. So [P] i =¿ P.

interp.c

```
[ 1 2 3 *] i          =>   1 6
 8 9[ pop 10 11] i    =>   8 10 11
```

i2 :

```
...  X [P1][P2] -> ...
```

```
== [dip]dip i;
```

inilib.joy

```
 4 5 6[ 10 *][ 10     =>   4 50 16
+] i2
 4 5 6[ *][ +] i2     =>   26
```

infra :

```
L [P] -> L
```

Using list L as stack, executes P and returns a new list. The first element of L is used as the top of stack, and after execution of P the top of stack becomes the first element of new L.

interp.c

```
[ 2 3 4 5][ *]        =>  [ 6 4 5]
infra
```

some :

```
A [P:test] -> B
```

Applies test to members of aggregate A and returns true if some (that is one or more) pass, false if not.

interp.c

```
[ 1 2 3][ odd]        =>   true
some
[ 2 4 6][ odd]        =>   false
some
```

x :

```
[P] -> ...
```

Executes P without popping [P]. So [P] x == [P] P.

interp.c

```
 1 2[ cons] x        =>   1[ 2 cons]
 1 2 3[ dup] x       =>   1 2 3[ dup][ dup]
```

8.8 Apply: app, nullary, unary, binary,...

app1	`X [P] -> X`
app11	`X X [P] -> X`
app12	`X Y Z [P] -> X X`
app2	`X X [P] -> X X`
app3	`X X X [P] -> X X X`
app4	`X X X X [P] -> X X X X`
binary	`X Y [P] -> Z`

nullary	`[P] -> X`
nullary2	`X Y [P] -> X Y V W`
ternary	`X Y Z [P] -> X`
unary	`X [P] -> Y`
unary2	`X Y [P] -> X Y`
unary3	`X Y Z [P] -> X Y Z`
unary4	`X X X X [P] -> X X X X`

app1 :

`X [P] -> X`

Executes P, pushes result new X on stack without old X.

Identical to i, except app1 expects 2 parameters while i expects only 1 on stack.

interp.c

```
 100 5 1[ * +]       =>   105
app1
 100 5 1[ * +] i     =>   105
 100 5 1[ * +]       =>   100 5 105
unary
```

app11 :

`X X [P] -> X`

Executes P, pushes result on stack.

interp.c

```
 100 5 1[] app11     =>   100 1
 100 5 1[ 9] app11   =>   100 5 9
 100 5 1[ 9 99]      =>   100 5 1 99
app11
 100 5 1[ 1 +]       =>   100 2
app11
 9 100 5 1[ * +]     =>   105
app11
 100 5 1[ * +] i     =>   105
 100 5 1[ * +]       =>   100 5 105
unary
```

app12 :

```
X Y Z [P] -> X X
```

Executes P twice with Y and Z returns 2 values.

interp.c

```
 1 2 3[] app12       =>   2 3
 1 2 3[ 999] app12   =>   999 999
 1 2 3[ 99 999]      =>   999 999
app12
 10 2 3[ 10 +]       =>   12 13
app12
 10 2 3[ +] app12    =>   12 13
 9 100 5 2[ * +]     =>   9 509 209
app12
 1                   =>   1
 9 100 5 2[ *][ +]   =>   9 100 5 10 7
cleave
 9 100 5 2[ * +] i   =>   9 110
 9 100 5 1[ * +]     =>   9 100 5 105
unary
 1 2 3[ 9] binary    =>   1 9
 1 2 3[ + +]         =>   1 6
binary
 1 2 3[] binary      =>   1 3
 1 2 3[ 99 999]      =>   1 999
binary
```

app2 :

```
X X [P] -> X X
```

```
== unary2;
```

Obsolescent.

interp.c

app3 :

```
X X X [P] -> X X X
```

```
== unary3;
```

Obsolescent.

interp.c

app4 :

```
X X X X [P] -> X X X X
```

```
== unary4;
```

Obsolescent.

interp.c

binary :

```
X Y [P] -> Z
```

Executes P which leaves Z on top of the stack. No matter how many parameters this consumes, exactly two are removed from the stack.

interp.c

```
 1 2 3[ 9] binary    =>   1 9
 1 2 3[ + +]         =>   1 6
binary
 1 2 3[] binary      =>   1 3
 1 2 3[ 99 999]      =>   1 999
binary
```

nullary :

```
[P] -> X
```

Executes P which leaves X on top of the stack. No matter how many parameters this consumes none are removed from the stack.

interp.c

```
 1 2 3[ +] nullary   =>   1 2 3 5
 1 2 3[] nullary     =>   1 2 3 3
 1 2 3[ 7 8 9]       =>   1 2 3 9
nullary
```

nullary2 :

```
X Y [P] -> X Y V W
```

```
== [ nullary] cons dup i2 swapd ;
```

inilib.joy

```
 2 3[ 10 +]          =>   2 3 12 13
nullary2
```

ternary :

```
X Y Z [P] -> X
```

Executes P which leaves new X on top of the stack. No matter how many parameters this consumes, exactly three are removed from the stack.

interp.c

```
 1 2 3[ 9] ternary   =>   9
 1 2 3[ + +]         =>   6
ternary
 1 2 3[] ternary     =>   3
 1 2 3[ 99 999]      =>   999
ternary
```

unary :

```
X [P] -> Y
```

Executes P which leaves Y on top of the stack. No matter how many parameters this consumes exactly one is removed from the stack.

interp.c

```
 100 5 1[ * +]       =>   100 5 105
unary
 1 2 3[] unary       =>   1 2 3
 1 2 3[ 7 8 9]       =>   1 2 9
unary
```

unary2 :

```
X Y [P] -> X Y
```

Executes P twice, with X and Y on top of the stack returning new X and new Y.

No matter how many parameters both executions consume, exactly two are removed from the stack.

interp.c

```
 100 5 1 2[ * +]     =>   100 5 105 110
unary2
```

unary3 :

```
X Y Z [P] -> X Y Z
```

Executes P three times, with X Y and Z on top of the stack returning new X, new Y and new Z.

No matter how many parameters the executions consume, exactly tree are removed from the stack.

interp.c

```
 100 5 1 2 3[ * +]   =>   100 5 105 110 115
unary3
```

unary4 :

```
X X X X [P] -> X X X X
```

Executes P four times, with X:1-4 on top of the stack returning four new X.

No matter how many parameters the executions consume, exactly four are removed from the stack.

interp.c

```
 100 5 1 2 3 4[ *    =>   100 5 105 110 115 120
+] unary4
```

Chapter 9

Time and Date

clock	`-> I`
gmtime	`I -> L`
localtime	`I -> L`
localtime-strings	`-> L`
mktime	`L -> I`
months	`-> L`

now	`-> S`
show-todaynow	`->`
strftime	`L Str -> Str`
time	`-> I`
today	`-> Str`
weekdays	`-> L`

clock :

`-> I`

Pushes the integer value of current CPU usage in hundreds of a second.

interp.c

```
clock                    =>   680000
```

gmtime :

`I -> L`

Converts a time I into a list L representing universal time:

[year month day hour minute second isdst yearday weekday].

Month is 1 = January ... 12 = December.

isdst is true for daylight savings/summer time, otherwise false.

weekday is 0 = Monday ... 6 = Sunday.

interp.c

```
time gmtime          =>   [ 2003 3 25 21 3 26 false 83 2]
0 gmtime             =>   [ 1970 1 1 0 0 0 false 0 4]
```

localtime :

`I -> L`

Converts a time I into a list L representing local time:

[year month day hour minute second isdst yearday weekday].

Month is 1 = January ... 12 = December.

isdst is true for daylight savings/summer time, otherwise false.

weekday is 0 = Monday ... 6 = Sunday.

interp.c

```
time localtime       =>   [ 2003 3 25 22 3 26 false 83 2]
```

localtime-strings :

`-> L`

Push a list with current time as follows:

[year month day hour minute second isdst year-day weekday].

inilib.joy

```
localtime-strings    =>   [ "2003" "MAR" "25" "22" "03" "26" "false"
                          "00083" "Tuesday"]
```

mktime :

`L -> I`

Converts a list L representing local time into a time I. L has to be in the format generated by localtime.

interp.c

```
[ 2003 3 20 8 20     =>    1048144800
0] mktime
[ 2003 1 20 8        =>   [ 2003 2 20 8 20 0 false 50 4]
20 0] mktime 31
24 60 60 * * * +
localtime
```

months :

`-> L`

`== [ "JAN" "FEB" "MAR" "APR" "MAY" "JUN" "JUL" "AUG" "SEP" "OCT" "NOV" "DEC"] ;`

x

inilib.joy

now :

`-> S`

Push a string containing current time.

inilib.joy

```
now                 =>   "22:03:26"
```

show-todaynow :

`->`

`== today putchars space now putchars newline ;`

Print current time and date to std-output.

inilib.joy

strftime :

`L Str -> Str`

Formats a list L in the format of localtime or gmtime using a string and pushes the result as string.

interp.c

time :

`-> I`

Pushes the current time (in seconds since the Epoch).

interp.c

```
time                =>   1048626206
```

today :

```
-> Str
```

Push a string containing todays date.

inilib.joy

```
 today                =>   "Tuesday 25-MAR-03"
```

weekdays :

```
-> L
```

```
== [ "Monday" "Tuesday" "Wednesday" "Thursday" "Friday" "Saturday"
"Sunday"] ;
```

inilib.joy

Chapter 10

Format

format	`I C I1 I2 -> Str`
formatf	`F C I1 I2 -> Str`

strtod	`Str -> F`
strtol	`Str I -> I`

format :

`I C I1 I2 -> Str`

Result is the formatted version of integer I in mode C with maximum width I1 and minimum width I2.

Possible modes of C:

'd or 'i: decimal

'o: octal

'x or X: hex

interp.c

```
12 'd 1 1 format    =>   "12"
12 'd 8 2 format    =>   " 12"
12 'd 8 4 format    =>   " 0012"
```

formatf :

`F C I1 I2 -> Str`

Result is the formatted version of Float F in mode C with maximum width I1 and precision I2.

Possible modes of C:

'e or E: exponential

'f: fractional

'g or G: general

interp.c

```
 12.89 'e 10 4       =>    "1.2890e+01"
formatf
 12.89 'f 10 4       =>    " 12.8900"
formatf
 12.89 'g 10 4       =>    " 12.89"
formatf
```

strtod :

`Str -> F`

String Str is converted to float F.

interp.c

strtol :

`Str I -> I`

String Str is converted to an integer using base I. If I = 0 assumes base 10 but leading "0" means base 8 and leading "0x" means base 16.

interp.c

```
 "12" 3 strtol       =>    5
```

Chapter 11

Stdin and Stdout

ask	`Str -> X`
bell	`->`
get	`-> X`
newline	`->`
put	`X ->`
putch	`C|I ->`
putchars	`Str ->`

putlist	`L ->`
putln	`X ->`
putstrings	`L ->`
space	`->`
stderr	`-> STREAM`
stdin	`-> STREAM`
stdout	`-> STREAM`

ask :

`Str -> X`

`== 'Please ' putchars putchars newline get;`

inilib.joy

bell :

`->`

`== XXXX007 putch;`

inilib.joy

get :

`-> X`

Reads a factor from input and pushes it onto stack.

Note: While including files (`'filename.joy' include`) the input into the Joy system is turned to the specified file. In that case get takes its input from the same file.

interp.c

newline :

`->`

`== XXXXXX putch;`

inilib.joy

put :

`X ->`

Writes X to output, pops X off stack.

interp.c

putch :

`C|I ->`

Writes character C or whose ASCII is I to stdout.

interp.c

putchars :

`Str ->`

Write Str without quotes to stdout.

interp.c

putlist :

`L ->`

Print a list user-readable to stdout.

`[ [1 2 3] [4 5 6] [7 8 [1 2 3] 9] ] putlist`is printed as

```
[ [1 2 3]
  [4 5 6]
  [7 8 [1 2 3] 9] ]
```

seqlib.joy

putln :

`X ->`

`== put newline;`

inilib.joy

putstrings :

`L ->`

`== [ putchars] step ;`

Print a list of strings.

inilib.joy

space :

`->`

`== XXXXX032 putch;`

inilib.joy

stderr :

`-> STREAM`

Pushes the standard error stream.

interp.c

stdin :

`-> STREAM`

Pushes the standard input stream.

interp.c

stdout :

`-> STREAM`

Pushes the standard output stream.

interp.c

Chapter 12

Files and Streams

fclose	`STREAM ->`
feof	`STREAM -> STREAM B`
ferror	`STREAM -> STREAM B`
fflush	`STREAM -> STREAM`
fgetch	`STREAM -> STREAM C`
fgets	`STREAM -> STREAM Str`
file	`Str -> B`
fopen	`Str C -> STREAM`
fput	`STREAM X -> STREAM`

fputch	`STREAM C -> STREAM`
fputchars	`STREAM Str -> STREAM`
fputstring	`STREAM Str -> STREAM`
fread	`STREAM I -> STREAM L`
fremove	`Str -> B`
frename	`Str1 Str2 -> B`
fseek	`STREAM I1 I2 -> STREAM`
ftell	`STREAM -> STREAM I`
fwrite	`STREAM L -> STREAM`

fclose :

`STREAM ->`

The stream is closed and removed from the stack.

interp.c

feof :

`STREAM -> STREAM B`

Boolean B is the end-of-file status of stream.

interp.c

ferror :

`STREAM -> STREAM B`

B is the error status of stream.

interp.c

fflush :

`STREAM -> STREAM`

Flush stream forcing all buffered output to be written.

interp.c

fgetch :

`STREAM -> STREAM C`

C is the next available character from stream.

interp.c

fgets :

`STREAM -> STREAM Str`

Str is the next available line from stream.

interp.c

file :

`Str -> B`

Tests whether string Str is a file.

interp.c

fopen :

`Str C -> STREAM`

The file system object with pathname Str is opened with mode C (r w a etc.) and stream object STREAM is pushed.

If the open fails file:NULL is pushed.

interp.c

fput :

`STREAM X -> STREAM`

Writes X to stream.

interp.c

fputch :

`STREAM C -> STREAM`

The character C is written to the current position of stream.

interp.c

fputchars :

```
STREAM Str -> STREAM
```

The string Str is written without quotes to the current position of stream.

interp.c

fputstring :

```
STREAM Str -> STREAM
```

```
== fputchars;
```

Temporary alternative to fputchars.

interp.c

fread :

```
STREAM I -> STREAM L
```

I bytes are read from the current position of stream and returned as a list of integers.

interp.c

fremove :

```
Str -> B
```

The file system object with pathname Str is removed from the file system. B is a boolean indicating success or failure.

interp.c

frename :

```
Str1 Str2 -> B
```

The file system object with pathname Str1 is renamed to Str2. B is a boolean indicating success or failure.

interp.c

fseek :

```
STREAM I1 I2 -> STREAM
```

Stream is repositioned to position I1 relative to whence-point I2 where I2 = 0 1 2 for beginning current position end respectively?

interp.c

ftell :

`STREAM -> STREAM I`

I is the current position of stream.

interp.c

fwrite :

`STREAM L -> STREAM`

A list of integers is written as bytes to the current position of stream.

interp.c

Chapter 13

Joy

13.1 Help and Debug

__html_manual	->
__latex_manual	->
__manual_list	-> L
_help	->
autoput	-> I
echo	-> I
help	->

helpdetail	L ->
manual	->
setautoput	I ->
setecho	I ->
setundeferror	I ->
undeferror	-> I

__html_manual :

`->`

Writes manual of all Joy primitives to output file in HTML style.

interp.c

__latex_manual :

`->`

Writes manual of all Joy primitives in Latex to output file but without the head and tail.

interp.c

__manual_list :

`-> L`

Pushes a list L of lists (one per operator) of three documentation strings for all atoms:

[["name" "type" "description"]...]

interp.c

_help :

`->`

Lists all hidden symbols in library and then all hidden inbuilt symbols.

interp.c

autoput :

`-> I`

Pushes current value of flag for automatic output.

interp.c

echo :

`-> I`

Pushes value of echo flag.

interp.c

help :

`->`

Lists all defined symbols including those from library files. Then lists all primitives of raw Joy.

interp.c

helpdetail :

`L ->`

Gives brief help on each item of L.

interp.c

manual :

`->`

Writes manual of all Joy primitives to output file.

interp.c

setautoput :

`I ->`

Sets value of flag for automatic put to I. The automatic output is executed as soon as Joy is returning to its main execution loop.

0: none

1: execute one put

2: print the stack

interp.c

setecho :

`I ->`

Sets the value of echo flag for listing of source code lines at stdout while including new files. This results in a mix of code lines with the results these lines produce.

0: no echo

1: echo

2: echo with tab

3: echo with tab and linenumber.

interp.c

setundeferror :

`I ->`

Sets flag that controls behavior of undefined functions.

0: no error

1: error

interp.c

undeferror :

`-> I`

Pushes current value of undefined-is-error flag.

interp.c

13.2 Program: quit, abort, name, body, ...

abort	`->`
body	`Sym -> [P]`
gc	`->`
intern	`Str -> Sym`

name	`Sym -> Str`
quit	`->`
user	`X -> B`

abort :

`->`

Aborts the execution of current Joy program and returns to Joy main cycle.

interp.c

body :

`Sym -> [P]`

Quotation [P] is the body of user-defined symbol Sym.

interp.c

`[ qsort] first body`	`=>`	`[[ small][][ uncons[ >] split][ swapd cons concat] binrec]`

gc :

`->`

Initiates garbage collection.

interp.c

intern :

`Str -> Sym`

Pushes the item whose name is string Str.

interp.c

`"qsort" intern`	`=>`	`qsort`
`"+" intern`	`=>`	`+`
`1 2 "+" intern call`	`=>`	`3`

name :

`Sym -> Str`

Result is the type of Sym for literals, the name of Sym for atoms and definitions.

interp.c

```
[ qsort] first       =>   "qsort"
name
[ +] first name      =>   "+"
 17 name             =>   " integer type"
```

quit :

`->`

Exit from Joy.

interp.c

user :

`X -> B`

Tests whether X is a user-defined symbol.

interp.c

13.3 Joy - System

__dump	->
__memoryindex	->
__memorymax	->

__settracegc	I ->
__symtabindex	->
__symtabmax	->

__dump :

`->`

debugging only: pushes the dump as a list.

interp.c

__memoryindex :

`->`

Pushes current value of memory.

interp.c

```
__memoryindex       =>   2397917
```

__memorymax :

`->`

Pushes value of total size of memory.

interp.c

__settracegc :

`I ->`

Sets value of flag for tracing garbage collection to I (= 0..5).

0: no gc trace

1: a little gc trace

2: a lot gc trace

3-5: ...

interp.c

__symtabindex :

`->`

Pushes current size of the symbol table.

interp.c

```
__symtabindex       =>   1279
```

__symtabmax :

`->`

Pushes value of maximum size of the symbol table.

interp.c

13.4 Library Loading

AGGLIB	->	Str
INILIB	->	Str
NUMLIB	->	Str
SEQLIB	->	Str
_agglib	->	B
_inilib	->	B
_numlib	->	B

_seqlib	->	B
all-libload	->	
basic-libload	->	
include	Str	->
libload	Str	->
special-libload	->	
verbose	->	B

AGGLIB :

```
-> Str
== "agglib.joy - aggregate library " ;
```

agglib.joy

INILIB :

```
-> Str
== "inilib.joy - the initial library, assumed everywhere " ;
```

inilib.joy

NUMLIB :

```
-> Str
== "numlib.joy - numerical library " ;
```

numlib.joy

SEQLIB :

```
-> Str
== "seqlib.joy - sequence library, assumes agglib.joy " ;
```

seqlib.joy

_agglib :

```
-> B
== true ;
```

agglib.joy

_inilib :

`-> B`

inilib.joy

_numlib :

`-> B`

`== true ;`

numlib.joy

_seqlib :

`-> B`

seqlib.joy

all-libload :

`->`

`== basic-libload special-libload ;`

inilib.joy

basic-libload :

`->`

`== "agglib" libload "seqlib" libload "numlib" libload ;`

inilib.joy

include :

`Str ->`

Transfers input to file whose name is specified by Str. On end-of-file returns to previous input file.The specified filename has to provide a filename extension.

interp.c

libload :

`Str ->`

Str specifies a filename without filename extension. If there is no symbol 'XXXXXXfilename' defined, the specified file is included.

inilib.joy

special-libload :

```
->
```

```
== "mtrlib" libload "tutlib" libload "lazlib" libload "lsplib"
libload "symlib" libload ;
```

inilib.joy

verbose :

```
-> B
```

```
== false;
```

If verbose is defined as false, there comes no notice if a library has already been loaded.

inilib.joy

Chapter 14

System

argc	`-> I`
argv	`-> L`

getenv	`Str -> Str`
system	`Str:command -> Str`

argc :

`-> I`

Pushes the number of command line arguments. This is equivalent to 'argv size'.

interp.c

```
argc                    =>   1
```

argv :

`-> L`

Creates a list L containing the interpreter's command line arguments.

interp.c

```
argv                    =>   [ "/self/adds/joy/newjoy/joy"]
```

getenv :

`Str -> Str`

Retrieves the value of the environment variable specified by Str.

interp.c

system :

`Str:command -> Str`

Escapes to shell and executes Str. The string may cause execution of another program (command). When that has finished the process returns to Joy.

interp.c

Part IV

Definitions of Additional Libraries

Chapter 15

Typelib.joy

15.1 Stack

st_new	`-> []`
st_null	`L:stack -> L:stack B`
st_pop	`L:stack -> L:stack`

st_pull	`L:stack -> L:stack X`
st_push	`L X -> L`
st_top	`L:stack -> L:stack X`

st_new :

```
-> []
== [] ;
```

Push a new and empty Stack.

typlib.joy

```
st_new              =>  []
```

st_null :

```
L:stack -> L:stack B
== dup null ;
```

Tests, if L:stack is empty.

typlib.joy

```
st_new st_null      =>  [] true
```

st_pop :

```
L:stack -> L:stack
== [ null] [ "st_pop " _st_chk] [ rest] ifte ;
```

Removes the top element of L:stack.

typlib.joy

```
[ 1 2] st_pop        =>  [ 2]
```

st_pull :

```
L:stack -> L:stack X
== [ null] [ "st_pull " _st_chk] [ unswons] ifte ;
```

Removes the top element from L:stack and returns it at Joy stack.

typlib.joy

```
[ 1 2] st_pull       =>  [ 2] 1
```

st_push :

```
L X -> L
== swons ;
```

Push a new stack item.

typlib.joy

```
 st_new 1 st_push    =>  [ 1]
```

st_top :

```
L:stack -> L:stack X
== [ null] [ "st_top " _st_chk] [ dup first] ifte ;
```

Copies the top of L:stack to Joy stack.

typlib.joy

```
[ 1 2] st_top        =>  [ 1 2] 1
```

15.2 Dictionary

_d_sample	
d_add	`L L -> L`
d_differ	`L L -> L`
d_look	`L X -> L X`

d_new	`-> []`
d_null	`L -> B`
d_rem	`L X -> L`
d_union	`L L -> L`

_d_sample :

```
== [[ 1 "1"][ 2 "2"][ 3 "3"][ 4 "4"][ "fred" "FRED"]] ;
```

typlib.joy

d_add :

```
L L -> L
```

Add a new entry to dictionary.

typlib.joy

`d_new[ 1 "one"] d_add[ 2 "two"] d_add`	`=> [[ 1 "one"][ 2 "two"]]`

d_differ :

```
L L -> L
```

?

typlib.joy

`[[ 1 "one"][ 2 "two"]][[ 'a "A"][ 2 "two"]] d_differ`	`=> [[ 1 "one"][ 2 "two"]]`
`[[ 1 "one"][ 2 "two"]][[ 'a "A"][ 2 "two"]] d_differ`	`=> [[ 1 "one"][ 2 "two"]]`

d_look :

```
L X -> L X
```

Look up a value.

typlib.joy

`[[ 1 "one"][ 2 "two"]] 2 d_look`	=>	`[[ 1 "one"][ 2 "two"]][ 2 "two"]`
`[[ 1 "one"][ 2 "two"]] 3 d_look`	=>	`[[ 1 "one"][ 2 "two"]] "not found"`

d_new :

`-> []`

`== [] ;`

Push a new and empty Dictionary.

typlib.joy

`d_new`	=>	`[]`

d_null :

`L -> B`

`== null ;`

Test, if dictionary is empty.

typlib.joy

`d_new d_null`	=>	`true`

d_rem :

`L X -> L`

Remove a value from dictionary.

typlib.joy

`[[ 1 "one"][ 2 "two"]] 2 d_rem`	=>	`[[ 1 "one"]]`
`[[ 1 "one"][ 2 "two"]] 3 d_rem`	=>	`[[ 1 "one"][ 2 "two"]]`

d_union :

`L L -> L`

Unify two dictionaries.

typlib.joy

`[[ 1 "one"][ 2 "two"]][[ 'a "A"][ 2 "zwei"]] d_union`	=>	`[[ 1 "one"][ 2 "two"][ 'a "A"][ 2 "zwei"]]`

15.3 Queue

q_add	`L L X -> L L`
q_addl	`L L L -> L L`
q_front	`L L -> L L X`

q_new	`-> [] []`
q_null	`L L -> L L B`
q_rem	`L L ->`

q_add :

`L L X -> L L`

Add X to queue.

typlib.joy

```
[] [] 1 q_add 2 q_add      =>   [ 2 1] []
```

q_addl :

`L L L -> L L`

Add list to queue.

typlib.joy

```
[ 1 2] [] [ 33 44] q_addl  =>   [ 44 33 1 2] []
```

q_front :

`L L -> L L X`

Copy the first element of queue onto Joy stack.

typlib.joy

```
[ 1 2] [] q_front          =>   [] [ 2 1] 2
```

q_new :

`-> [] []`

`== [] [] ;`

Push a new and empty Queue.

typlib.joy

```
q_new                      =>   [] []
```

q_null :

`L L -> L L B`

Test, if queue is empty.

typlib.joy

```
[] [] q_null          =>   [] [] true
```

q_rem :

`L L ->`

Remove the first element from queue and push it onto Joy stack.

typlib.joy

```
[ 1 2] [] q_rem       =>   [] [ 1] 2
```

15.4 Big Set

bs_delete	
bs_differ	L1 L2 -> L
bs_insert	L X -> L

bs_member	L X -> B
bs_new	-> []
bs_union	L L -> L

bs_delete :

typlib.joy

bs_differ :

`L1 L2 -> L`

Result is big set L1 without members of big set L2.

typlib.joy

```
[ 1 2 3] [ 3]         =>   [ 1 2]
bs_differ
[ 1 2 3] [ 4]         =>   [ 1 2 3]
bs_differ
```

bs_insert :

```
L X -> L
```

Insert X in big set L.

typlib.joy

```
[ 1 2 9] "New"      =>  [ 1 2 9 "New"]
bs_insert
[ 1 2 9] 5          =>  [ 1 2 5 9]
bs_insert
```

bs_member :

```
L X -> B
```

Test, if X is member of big set L.

typlib.joy

```
[ 1 2 3 4] 4        =>   true
bs_member
[ 1 2 3 4] 5        =>   false
bs_member
```

bs_new :

```
-> []
```

```
== [] ;
```

Push a new and empty Big Set.

typlib.joy

```
 bs_new             =>  []
```

bs_union :

```
L L -> L
```

Unification of two big sets.

typlib.joy

```
[ 1 2][ 8 9]        =>  [ 1 2 8 9]
bs_union
```

15.5 Tree

_t_sample -> L t_add t_front t_new -> []	t_null t_rem t_reset

_t_sample :

```
-> L
== [ 1 20[ 3 40][ 5 60] 70[[[ 8]]]] ;
```

typlib.joy

t_add :

typlib.joy

t_front :

typlib.joy

t_new :

```
-> []
```

Push a new and empty Tree.

typlib.joy

```
t_new                    =>   []
```

t_null :

typlib.joy

t_rem :

typlib.joy

t_reset :

typlib.joy

15.6 Debug

fatal2 `Str Str ->`

fatal2 :

`Str Str ->`

`== putchars putchars newline abort ;`

Print two error messages and abort.

typlib.joy

Chapter 16

Some.joy

16.1 Stack Manipulation

nop	->
over	X Y -> X Y X
over2	X Y V W-> X Y V W X Y

overd	X Y Z -> X Y X Z
pop3	X X X ->
swap2	X Y V W -> V W X Y

nop :

```
->
```

```
== id ;
```

Does nothing.

some.joy

```
"X" nop                =>   "X"
```

over :

```
X Y -> X Y X
```

```
== dupd swap ;
```

some.joy

```
1 2 over               =>   1 2 1
```

over2 :

```
X Y V W-> X Y V W X Y
```

```
== [ dup2] dipd swap2 ;
```

some.joy

```
 1 2 'a 'b over2     =>   1 2 'a 'b 1 2
```

overd :

```
X Y Z -> X Y X Z
== [ over] dip ;
```

some.joy

```
 1 2 99 overd        =>   1 2 1 99
```

pop3 :

```
X X X ->
== pop2 pop ;
```

some.joy

```
 1 2 3 4 pop3        =>   1
```

swap2 :

```
X Y V W -> V W X Y
== rollupd rollup ;
```

some.joy

```
 1 2 'a 'b swap2     =>   'a 'b 1 2
```

16.2 Aggregates

concat3	`A A A -> A`
concatall	`A -> Str`
dequote	`[P] -> [P]`
docca	`[P] -> Str`
doccaif	`X -> Str`
isin	`X A -> B`

last	`A -> X`
pair	`X Y -> L`
resize	`Str I -> Str`
unzip	`A -> A A`
wrapconcat	`A A A -> A`
zipwith	`A A [P] -> A`

concat3 :

`A A A -> A`

`== concat concat ;`

some.joy

concatall :

`A -> Str`

`== "" [ concat] fold ;`

Concatall concats all strings of a list into one string.

Flatten concats all lists of a list into one list.

some.joy

```
[ " Max" "          =>   " Max Mueller Muenchen"
Mueller" "
Muenchen"]
concatall
[[ 1 2][ 3 4]]      =>  [ 1 2 3 4]
flatten
```

dequote :

`[P] -> [P]`

`== [] swap infra reverse ;`

Evaluates to the evaluation of P, returned as a new quotation.

some.joy

```
[ 16 2 + 20 4 +]     =>  [ 18 24]
dequote
[ 16 2 + 20 4 +] i   =>   18 24
[][ 16 2 + 20 4 +]   =>  [ 24 18]
infra
```

docca :

`[P] -> Str`

`== dequote concatall ;`

Evaluates to the string concatenation of the evaluation of P.

some.joy

```
[ "Max " "Mueller    =>    "Max Muell"
" 5 resize] docca
```

doccaif :

```
X -> Str
```

If X is a list docca it.

some.joy

```
[ "Max " "Mueller    =>    "Max Muell"
" 5 resize]
doccaif
 "Max Mueller"       =>    "Max Mueller"
doccaif
```

isin :

```
X A -> B
```

```
== swap [ equal] cons some ;
```

Test if X is in A. Works if X is an aggregate, too.

some.joy

```
 "word"[ "This" "is" "a" "word"] isin => true
 "word"[ "This" "is" "a" "sentence"] isin => false
 "word"[ "This" "is" "a" "word"] in => false
```

last :

```
A -> X
```

Returns the last element of A.

some.joy

```
[ 1 2 3] last       =>    3
```

pair :

```
X Y -> L
```

```
== [] cons cons ;
```

some.joy

```
 "Frank" 17 pair     =>   [ "Frank" 17]
```

resize :

`Str I -> Str`

Resize Str to length of I. That is cut down if Str is longer and append spaces if Str is shorter than I characters.

some.joy

```
 "Mueller" 4        =>   "Muel"
resize
```

unzip :

`A -> A A`

`== reverse [[][]] dip [ unpair swons2] step ;`

Unzip a list into two.

some.joy

```
[[ 1 2][ 11 22][     =>  [ 1 11 111][ 2 22 222]
111 222]] unzip
```

wrapconcat :

`A A A -> A`

`== [ swoncat] dip concat ;`

some.joy

```
 "center" "[" "]"   =>   "[center]"
wrapconcat
```

zipwith :

`A A [P] -> A`

```
== [[ null2][ pop2[]][ uncons2]] dip [ dip cons] cons linrec
;
```

Zip 2 aggregates, combining by P.

some.joy

```
[ 1 2 3][ 10 20      =>  [ 11 22 33]
30][ +] zipwith
```

16.3 Numerics

ipow	`N I -> N`

ipow :

```
N I -> N
== 1 rotate [ *] cons times ;
```

I times N * N, that is N raised to I-th power.

some.joy

```
 2 3 ipow           =>   8
```

16.4 Combinators

apps	`[ [P:app-1]...[P:app-n] ] I:-ary -> X1 ...  Xn`
b	`[P] [P] -> ...`
dipdd	`X Y Y Y [P] -> P(X) Y Y Y`
dudip	`X [P] -> P(X) X`
fold-andconds	`[ [P:app-1]..[P:app-n] ] I:n-ary -> X`
fold-listconcat	`[ [P:app-1]..[P:app-n] ] I:n-ary -> X`
fold-orconds	`[ [P:app-1]..[P:app-n] ] I:n-ary -> X`
fold-strconcat	`[ [P:app-1]..[P:app-n] ] I:-ary -> X`
foldapps	`[ [P:app-1]..[P:app-n] ] I:n-ary X:init [P:fold] -> X`
intersect	`A I:Index -> A A`
onitem	`A [P] I:Index -> A`
sdip	`X Y [P] -> P(Y) X`
step2	`A A [P] -> ...`
twice	`X [P] -> P(P(X))`

apps :

```
[ [P:app-1]...[P:app-n] ] I:-ary -> X1 ...  Xn
== [[ nullary] map] dip [ popd] times reverse [] step ;
```

First Parameter is a list of quoted programs. Each quoted program is applied to the same stack and the results of these applications are returned.

No matter how many parameters these operations consume, exactly I:-ary are removed from incoming stack.

some.joy

```
 "XY" 7 9[[ *][ +][ -]] 2 apps => "XY" -2 16 63
```

```
 "XY" 7 9[[ pop2][ "Lost" "Result"]] 0 apps => "XY" 7 9 "Result"
"XY"
```

```
 "XY" 7 9[ 1 +][[ *][ +][ -]] construct => "XY" 7 9 70 17 -3
```

b :

```
[P] [P] -> ...
```

```
== [ i] dip i ;
```

Executes both quoted programs.

some.joy

```
 16 2 4[ *][ /] b    =>   2
```

dipdd :

```
X Y Y Y [P] -> P(X) Y Y Y
```

```
== [ dipd] cons dip ;
```

some.joy

```
 17 1 2 3[ 100 *]    =>   1700 1 2 3
dipdd
```

dudip :

```
X [P] -> P(X) X
```

```
== [ dup] dip dip ;
```

some.joy

```
 17[ 10 *] dudip     =>   170 17
```

fold-andconds :

```
[ [P:app-1]..[P:app-n] ] I:n-ary -> X
```

```
== true [ and] foldapps ;
```

First parameter is a list of quoted programs. Each quoted program is applied to the same stack and expected to return a truth value. If all values are true, true is returned.

No matter how many parameters these operations consume, exactly I:-ary are removed from incoming stack.

some.joy

```
[ "Max" "Mueller" 37 6500 3][[ third 40 <][ fourth 5000 >][
fifth 3 =]] 1 fold-andconds => true
```

fold-listconcat :

```
[ [P:app-1]..[P:app-n] ] I:n-ary -> X
```

```
== [] [ concat] foldapps ;
```

some.joy

fold-orconds :

```
[ [P:app-1]..[P:app-n] ] I:n-ary -> X
```

```
== false [ or] foldapps ;
```

some.joy

fold-strconcat :

```
[ [P:app-1]..[P:app-n] ] I:-ary -> X
```

```
== "" [ concat] foldapps ;
```

First parameter is a list of quoted programs. Each quoted program is applied to the same stack and expected to return a string. The result strings are concatenated to one string and returned.

No matter how many parameters these operations consume, exactly I:-ary are removed from incoming stack.

some.joy

```
 "Muenchen"[ " Max" "Mueller"][[ first][ " "][ second][ ", "][
pop]] 2 fold-strconcat => " Max Mueller, Muenchen"
```

foldapps :

```
[ [P:app-1]..[P:app-n] ] I:n-ary X:init [P:fold] -> X
```

```
== [[[ nullary] map] dip[ popd] times] dip2 fold ;
```

some.joy

intersect :

`A I:Index -> A A`

Intersect aggregate A at position I into two aggs.

some.joy

```
[ 1 2 3 4] 2          =>  [ 1 2][ 3 4]
intersect
[ 1 2 3 4] 0          =>  [][ 1 2 3 4]
intersect
[ 1 2 3 4] 30         =>  [ 1 2 3 4][]
intersect
```

onitem :

`A [P] I:Index -> A`

Apply [P] to item I of agg A.

some.joy

```
[ 1 2 3 4][ 200 +]   =>  [ 201 2 3 4]
0 onitem
[ 1 2 3 4][ 200 +]   =>  [ 1 2 3 204]
3 onitem
[ 1 2 3 4][ pop      =>  [ 1 2 999 4]
999] 2 onitem
```

sdip :

`X Y [P] -> P(Y) X`

`== [ swap] dip dip ;`

some.joy

```
 1 2 3[ 10 * +]      =>   31 2
sdip
```

step2 :

`A A [P] -> ...`

`== [[ dup] dip] swoncat [ step pop] cons cons step ;`

some.joy

`[][ 1 2 3][ 4 5 6 0][ pair swap[ swons] dip] step2`	=>	`[[ 3 0][ 3 6][ 3 5][ 3 4][ 2 0][ 2 6][ 2 5][ 2 4][ 1 0][ 1 6][ 1 5][ 1 4]]`

twice :

`X [P] -> P(P(X))`

`== dup dip i ;`

some.joy

`3[ 2 *] twice`	=>	`12`

16.5 Stdin and Stdout

newputline `Str ->`	putline `Str ->`

newputline :

`Str ->`

`== newline putline ;`

Print newline at stdout followed by Str and newline.

some.joy

putline :

`Str ->`

`== putchars newline ;`

Print Str at stdout followed by newline.

some.joy

16.6 Files and Streams

get-file-contents	`Str:Filename -> Str:File-contents`
write-file-contents	`Str:Contents Str:Filename ->`

get-file-contents :

`Str:Filename -> Str:File-contents`

Open Filename and read it.

some.joy

write-file-contents :

`Str:Contents Str:Filename ->`

Open Filename and write Contents to it.

some.joy

16.7 Joy

calld `Symbol -> ...`	needed-time `[P] -> [...] F`

calld :

`Symbol -> ...`

`== [ call] dip ;`

some.joy

needed-time :

`[P] -> [...] F`

`== clock swap [] swap infra clock rolldown - 1000.00 / ;`

Evaluates to i(P), returned in a list and the time needed for the operation in msec.

some.joy

```
[ 1 1 +]              =>  [ 2] 0.00
needed-time
```

16.8 Debug

cp	`->`
cpbinrec	`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`
cpifte	`[P:condition] [P:if-true] [P:if-false] -> ...`
error	`X Str ->`
ncp	`Str ->`
ntp	`Str ->`
tp	`->`
tpbinrec	`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`
tpifte	`[P:if] [P:then] [P:else] -> ...`

cp :

`->`

Control Point: Print stack at stdout and wait for user command.

Available commands are:

v: Print stack vertical.

h: Print stack in a line.

j: Read Joy command from stdin and evaluate it.

c: Continue with program execution.

q: Quit program execution.

some.joy

cpbinrec :

`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`

Inserts Control Point at beginning and end of each quoted program and after that executes binrec.

some.joy

cpifte :

`[P:condition] [P:if-true] [P:if-false] -> ...`

Inserts a control point at beginning and end of all three quoted programs and after that executes ifte.

some.joy

error :

`X Str ->`

Print X and Str at stdout and abort.

some.joy

ncp :

`Str ->`

`== putchars cp ;`

Named Control Point: same as cp, additionally print Str.

some.joy

ntp :

`Str ->`

`== putchars tp ;`

Named Trace Point: same as tp, additionally print Str.

some.joy

tp :

`->`

Trace Point: Print stack at stdout and continue.

some.joy

tpbinrec :

`[P:condition] [P:if-true] [P:R1] [P:R2] -> ...`

Inserts Trace Point at beginning and end of each quoted program and after that executes binrec.

some.joy

tpifte :

`[P:if] [P:then] [P:else] -> ...`

cpifte inserts a trace point at beginning and end of all three quoted programs and executes ifte.

some.joy

Index