Lit	Bin	Desc	Arg Types	Autos	Example
	1	integer	{int}		3 → 3
"	2	string	{str}		"hi\n" → "hi\n"
'	d2	char	{chr}		'b' → 'b'
$	3	1st arg			;1;2;3 $ → 1,2,3,3
@	4	2nd arg			;1;2;3 @ → 1,2,3,2
_	5	3rd arg			;1;2;3 _ → 1,2,3,1
;	6	save	any		+ ;3 $ → 6
:	7	append	any∗ any∗	[]	:"abc" "def" → "abcdef"
]	8	max	num >num	0	]4 5 → 5
+	8	add	num vec	1 1	+2 1 → 3
[	a	min	int >num		[4 5 → 4
∗	a	multiply	int vec	-1 2	∗7 6 → 42
−	9	subtract	num num	1 1	− 5 3 → 2
/	b	divide	num num	2	/7 2 → 3
%	c	modulus	num num	2	%7 2 → 1
^	e	pow (minus is nth root)	int int	10 2	^2 8 → 256
+	8	sum	[int]		+,3 → 6
+	8	concat	[[∗]]		+.,3,$ → [1,1,2,1,2,3]
.	c	map	[∗] fn		."abc"+1$ → "bcd"
|	9	filter	[∗] reqfn	not◦	|,5%$2 → [1,3,5]
!	9	zip with	[∗] const zipop		! ,3"abc" + → "bdf"
/	a	foldr1	[∗] reqfn		/,3+@$ → 6
/	a	foldr	[∗] const fn	*tuple*	/,3 1 +@$ → 7
\	b	reverse	[∗]		\,3 → [3,2,1]
,	d	length	[∗]		,"asdf" → 4
<	b	take (`< also drop)	num [∗]	while	<3"asdfg" → "asd"
>	c	drop (`> also take)	int [∗]	while	>3,5 → [4,5]
,	d	range 1.. (`, is 0...)	num	∞	,3 → [1,2,3]
^	e	replicate	int [∗]|chr	∞	^3"ab" → "ababab"
=	9	subscript (wrapped)	num [∗]	*nowrap*	=2 "asdf" → 's'
?	f	index (or 0 if not found)	[a] a	by	? "abc" 'b' → 2
−	f	diff	[a] [a]		−"abcd" "bd" → "ac"
%	8	split (remove empties)	str str|chr	words	%"a b c" " " → ["a","b","c"]
∗	8	join	str [∗]		∗" ",3 → "1 2 3"
&	8	justify	str int vec	*center*	& " " 4 "hi" → " hi"
\	a	char class?	chr chclass		|"a.c d" \$a → "acd"
o	e	ord	chr		o'd' → 100
?	f	if/else	num fn fn	default	? 0 "T" "F" → "F"
`/	9 d	divmod	num num	2	`/7 2 $ → 3,1
`%	b d	moddiv	num num	2	`%7 2 $ → 1,3
`r	f 1	read int	str		`r"12" → 12
?,	fd	if/else (lazy list)	[∗] fn fn	default	?, "cond" 1 0 → 1
>>	c0	tail	[∗]		>>,5 → [2,3,4,5]
<<	b0	init	[∗]		<<"asdf" → "asd"
`(	e 1	uncons	[∗]		`(,3 $ → 1,[2,3]

Lit	Bin	Desc	Arg Types	Autos	Example
`)	e 2	swapped uncons	[*]		`) ,3 $ → [2,3],1
`/	be	chunks of	num [*]	2	`/2,5 → [[1,2],[3,4],[5]]
`\	de	n chunks	int [*]	2	`\ 2 ,6 → [[1,2,3],[4,5,6]]
`%	bc	step	num [*]	2	`%2,5 → [1,3,5]
`?	e a	find indices [by]	[*] fn|const	not◦	`? "a b" \$a → [1,3]
`=	b9	chunk by	[*] reqfn		`= ,5 /$2 → [[1],[2,3],[4,5]]
=~	bc	group by (also sorts)	[*] reqfn	*nosort*	=~ "cabcb" $ → ["a","bb","cc"]
or	bc	or	[*] const	*tbd*	or"" "b" or "a" "c" → "b","a"
`<	e d	sort	[*]		`<"asdf" → "adfs"
`'	e 3	transpose	[*]		`' "hi""yo" → ["hy","io"]
=\	dc	scanl1	[*] reqfn		=\,3+*2$@ → [1,5,11]
=\	dc	scanl	[*] const fn	*tuple*	=\,3 0 +@$ → [0,1,3,6]
`\	e c	special scans	[*] foldop		`\ ,4 + → [1,3,6,10]
`/	e b	special folds	[*] foldop		`/ "asdf"] → 's'
`*	90	product	[int]		`*,4 → 24
`_	cc	subsequences	int [*]	2	`_ 2 "abc" → ["ab","ac","bc"]
`*	90	nary cartesian product	[[*]]		*" "`*"ab""cd" → "ac ad bc bd"
``p	b80	permutations	[*]		``p "ab" → ["ab","ba"]
`:	e 8	list of 2 lists	[a] [a]		`: ,2 ,1 → [[1,2],[1]]
`-	e 8	list difference [by]	[*] fn? any	*uniq*	`- "aabd" 'a' → "abd"
`&	e 5	list intersection [by]	[*] fn? any	*uniq*	`& "abaccd" "aabce" → "abac"
`|	e 6	list union [by]	[*] fn? any	*uniq*	`| "abccd" "aabce" → "abccdae"
`^	e 7	list xor [by]	[*] fn? any	*uniq*	`^ "aabce" "abbde" → "acbd"
`$	e 4	uniq	[*]		`$ "bcba" → "bca"
!=	c0	abs diff	num >num	0	!= 3 5 → 2
`&	c0	bit intersection	num num	-2	`& 6 3 → 2
`|	b0	bit union	num >num	1	`| 3 6 → 7
`^	b0	bit xor	num num	1	`^ 6 3 → 5
|	9	partition	[*] ~ ~ fn	not◦	|,5~~%$2 $ → [1,3,5],[2,4]
%~	e 0	split by	[*] fn	not◦	%~"a b"$ → [("a",""),("b"," ")]
`%	f1	split list (keep empties)	[*] any	default	`% ,5 :3 4 → [[1,2],[5]]
-~	90	strip	str		-~ " bcd\n\n" → "bcd"
`$	d7	signum	num		`$ -2 `$ 0 `$ 2 → -1,0,1
`)	c 2	to uppercase	chr		`) 'a' → 'A'
`(	c 7	to lowercase	chr		`('A' → 'a'
ch	dd	chr	int	256	ch 100 → 'd'
`p	f1	int to str	num		`p 5 → "5"
``@	d70	to bits	num		``@ 10 → [1,0,1,0]
hex	b02	to/from hex	any		hex 31 → "1f"
`D	800	to base from data	{int}		`D 2 10 → [1,0,1,0]
`@	87	to base	int num	10 tbd	`@ 2 10 → [1,0,1,0]
`@	b0	from base	num [*]	10	`@ 2 :1 :0 :1 0 → 10
`.	b2	iterate while uniq	any* fn	*inf*	`. 10 %+1$3 → [10,2,0,1]
.~~	900	append until null	any* fn		.~~,3,-/$$1 → [1,2,3,1,2,1]
;~	60	save fn	any* reqfn		;~2+1$ $4 → 3,5

Lit	Bin	Desc	Arg Types	Autos	Example
==	60	equal?	any* const		== 1 2 == 1 1 → 0,1
`;	66	recursion (alt name ``;)	any* fn		`; 5 $ 1 *@-$~$ → 120
`#	f0	hashmod	any {int}	*nosalt*	."Fizz""Buzz" `# $ 6 1a → [3,5]
fsb		find salt by	[*] int int fn	∞ not◦	fsb"Fizz""Buzz"6~==$:3 5 → "1a"
fs		find salt	[*] int int [*]	∞	fs "Fizz""Buzz"6~ :3 5 → "1a"
pt		debug arg type	any		pt 5 → error"Integer"
p		show	any		p"a" → "\"a\""
ct		debug context types			;5 ct → error"$ :: Integer ..."
let		let statement			let x +5 4 *x x → 81
\		lambda (only in fns)			/,3 \e a +a e → 6
sets		name extras (; ok)			`/ 10 3 sets a a → 3,1
error	fd02	error	any		error +2 1 → error "3"

Legend:

symbol	meaning
{type}	parse a value of that type (not an expression, saves a nibble though)
any*	any type or ~ for a tuple
>type	this argument must be longer in length than the preceding (for commutative extensions)
vec	same as "any" but used to denote that it will vectorize if a list is given
num	int|chr
[*]	list of any type
[a]	list of type variable "a"
reqfn	fn but its argument must be used
const	fn but its argument must not be used (for extensions)
fn?	auto ~ means to take an extra fn argument here
itatlic	auto ~ specifies "option present" but does not replace the arg

Special args:

chclass		zipop		foldop	
a	isAlpha	]	max		
A	not.isAlpha	[	min		
n	isAlphaNum	+	add		
N	not.isAlphaNum	*	mult		
s	isSpace	-	sub		
S	not.isSpace	/	div		
l	isLower	%	mod		
L	not.isLower	^	pow		
u	isUpper	:	cons		
U	not.isUpper	~	by	|	or
p	isPrint	,	make tuple	&	and
P	not.isPrint	!	abs diff	;	rev cons
d	isDigit	=	subscript	>	max by fn
D	not.isDigit	?	index	<	min by fn
$	isSym				
!	not.isSym				

Inputs:

sym*	type	name	default
$	int	fstInt	100
@	str	fstLine	printable chars
_	[int]	ints	[]
;$	int	sndInt	1000
;@	[str]	allLines	[]
;_	str	allInput	""
;;$	[[int]]	intMatrix	[]
;;@	str	sndLine	""

* command args can be of any type and precede these (use Haskell syntax)

Implicit Ops (top level multiple expressions):

1st type	arg used	meaning
int	$ or @	range
list	@	foldl
list	$	map
~		encode data