# 1 Standard Library

The Standard Library, called `stdlib`, is always imported in every *V* program. It provides basic functions for a number of use cases, ranging from numerical operations to function manipulation.

Some basic language features, such as list comprehensions and ranges, depend on the existence of the `stdlib`. This means that, while it is possible to create programs without importing the `stdlib`, doing so will most likely break any existing program.

## 1.1 Operations on Basic Values

### 1.1.1 Operations on All Values

The operations below are helper functions, designed to allow cleaner code.

**id** :: $a \to a$

Identity function.

**const** :: $a \to b \to a$

Always returns the first parameter it is passed.

### 1.1.2 Operations on Numbers

The 4 basic operations (addition, subtraction, multiplication and division) are built into the language. Other operations must be defined in terms of these.

One important thing to note is that the unary negation operator (`-`) is tightly coupled with the `negate` function defined in the `stdlib`. While the operator is defined inside the language, it depends on the presence of the `stdlib` to function.

**remainder** :: $Int \to Int \to Int$

Integer remainder, satisfying:

```
(x / y) * y + (remainder x y) = x
```

**(%) :: $Int \to Int \to Int$** | left-associative, priority 8 |

Infix version of `remainder`

**negate** :: $Int \to Int$

Unary negation, satisfying:

```
x + (negate x) = 0
```

**abs** :: *Int* → *Int*

Absolute value

### 1.1.3 Operations on Booleans

Below are all the operations on booleans defined in the Standard Library.

**and** :: *Bool* → *Bool* → *Bool*

Boolean "AND"

**(&&) :: *Bool* → *Bool* → *Bool*** | right-associative, priority 3 |

Infix version of `and`

**or** :: *Bool* → *Bool* → *Bool*

Boolean "OR"

**(||) :: *Bool* → *Bool* → *Bool*** | right-associative, priority 2 |

Infix version of `or`

**not** :: *Bool* → *Bool*

Boolean "not"

```
not True = False
not False = True
```

**xor** :: *Bool* → *Bool* → *Bool*

Boolean "xor"

```
xor True True = False
xor True False = True
xor True False = True
xor False False = False
```

### 1.1.4 Operations on Functions

Basic manipulation of functions and application. Most of the usefulness of these functions come from their infix versions. They allow more compact and easier to read code

to be written, mainly reducing the need for parentheses.

**flip** :: $(a \to b \to c) \to b \to a \to c$

`flip f` takes its first two arguments in reverse order of `f`.

```
flip f x y = f y x
```

**apply** :: $(a \to b) \to a \to b$

This function simply applies its second argument to its first. While this seems redundant (after all, `apply f x` is the same as `f x`), it can be used higher order situations.

**($ )** :: $(a \to b) \to a \to b$ | right-associative, priority 1 |

Infix version of `apply`. While it has the same funcionality as normal function application, it is right-associative with the lowest possible priority.

In some situations, this allows parentheses to be omitted.

```
f $ g $ h x = f (g (h x))
```

**compose** :: $(b \to c) \to (a \to b) \to a \to c$

Function composition. Applies the third argument to the second one, applying the resulting value to the first argument.

```
compose f g x = f (g x)
```

**(.)** :: $(b \to c) \to (a \to b) \to a \to c$ | right-associative, priority 9 |

Infix version of `compose`.
Can be used with `$` to reduce the number of parentheses needed.

```
f .  g .  h $ x = f (g (h x))
```

### 1.1.5 Operations on Tuples

The stdlib also provides basic functions for manipulating tuples with 2 components. For larger tuples, it is necessary to create custom functions.

**fst** :: $(a, b) \to a$

Returns the first component of a pair.

**snd** :: $(a, b) \to b$

Returns the second component of a pair.

**swap** :: $(a, b) \to (b, a)$

Swap the components of a pair.

### 1.1.6 Operations on Records

The functions below are used with record accessors (`#label`) to get, set and change individual fields in a record. While the functions themselves have a more generic type and can, therefore, be used in more circumstances, they were created with records in mind.

**get** :: $(a \to b \to (c, d)) \to b \to c$

Returns the value of a field in a record.

```
get #label record
```

**set** :: $(a \to b \to (c, d)) \to a \to b \to d$

Returns the inputed record, modifying a single field value.

```
set #label value record
```

**modify** :: $(a \to b \to (c, d)) \to (c \to a) \to b \to d$

Returns the inputed record, modifying a single field value by applying the old value to the specified function.

```
modify #label function record
```

## 1.2 Operations on Lists

### 1.2.1 Basic Operations

Basic functions to aid in using lists.

**head** :: $[a] \to a$

Returns the first element of a list, which must have at least one element.

**last** :: $[a] \to a$

Returns the last element of a list, which must have at least one element.

**tail** :: $[a] \rightarrow [a]$

Removes the first element of a list, which must have at least one element.

**init** :: $[a] \rightarrow [a]$

Removes the last element of a list, which must have at least one element.

**tail** :: $[a] \rightarrow [a]$

Removes the first element of a list, which must have at least one element.

**empty?** :: $[a] \rightarrow \mathit{Bool}$

Returns `True` if the list is empty, and `False` otherwise.

**length** :: $[a] \rightarrow \mathit{Int}$

Returns the number of elements in the list.

**append** :: $a \rightarrow [a] \rightarrow [a]$

Adds an element to the end of a list.

**concat** :: $[a] \rightarrow [a] \rightarrow [a]$

Appends two lists, maintaning order.

(@) **::** $[a] \rightarrow [a] \rightarrow [a]$ | right-associative, priority 5 |

Infix version of `concat`.

### 1.2.2 Generation Operations

These operations create lists based on input values.

**range** :: $\mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow [\mathit{Int}]$

`range start finish increment` generates a list of the form
[`start`, `start` + `increment`, `start` + 2 * `increment`, ..., `n`], where

$\texttt{increment} > 0 \implies \texttt{n} \leq \texttt{finish}$
$\texttt{increment} < 0 \implies \texttt{n} \geq \texttt{finish}$

### 1.2.3 Transformation Operations

These operations transform a list, altering its elements, their order, or both.

**reverse** :: $[a] \to [a]$

Returns the elements of the input in reverse order.

**map** :: $(a \to b) \to [a] \to [b]$

`map f ls` returns a list by applying the function `f` to each element of the list `ls`.

### 1.2.4 Reduction Operations

These operations take a list and reduce it to a simple value.

**fold** :: $(b \to a \to b) \to b \to [a] \to b$

`fold f acc ls` reduces the list using the function `f`, applying it to an accumulator (`acc`) and each element of the list, from left to right.

**reduce** :: $(a \to a \to a) \to [a] \to a$

The same as `fold`, but using the first element of the list as the `acc`

**all** :: $(a \to \mathit{Bool}) \to [a] \to \mathit{Bool}$

Checks whether all elements of a list satisfy a predicate. An empty list returns true.

**any** :: $(a \to \mathit{Bool}) \to [a] \to \mathit{Bool}$

Checks whether any elements of a list satisfy a predicate. An empty list returns false.

**maximum** :: $\mathit{Orderable}\ a \Rightarrow [a] \to a$

Returns the largest element of the list.

**minimum** :: *Orderable a => [a] → a*

Returns the smallest element of the list.

### 1.2.5 Sublist Operations

These operations return smaller segments of an existing list.

**take** :: *Int → [a] → [a]*

`take n ls` returns the first `n` elements of `ls`.

**drop** :: *Int → [a] → [a]*

`drop n ls` returns the list resulting from removing the first `n` elements of `ls`.

**takeWhile** :: *(a → Bool) → [a] → [a]*

`takeWhile p ls` returns the longest prefix of `ls` such that every element satisfies `p`.

**dropWhile** :: *(a → Bool) → [a] → [a]*

`dropWhile p ls` returns the suffix that remains after `takeWhile p ls`.

**sublist** :: *Int → Int → [a] → [a]*

`sublist start length ls` drops the first `start` elements of `ls`, and then takes the first `length` elements of the resulting list.

### 1.2.6 Search Operations

These operations search for specific elements in a list.

**exists** :: *Equatable a => a → [a] → Bool*

Tests whether the given element exists in the list.

**filter** :: *(a → Bool) → [a] → [a]*

`filter p ls` returns a sublist of `ls` such that every element satisfies `p`.

### 1.2.7 Indexing Operations

Manipulate a list through the index of its elements

**indexOf** :: *Equatable a => a → [a] → Int*

`indexOf t ls` returns the index of the first occurrence of `t` in `ls`. If the element does not occur, returns `-1`.

**nth** :: *Int → [a] → a*

`nth n ls` returns the element of `ls` at position `n`. If `n` is negative or larger than `length ls`, an exception is raised.

**(!!) :: [a] → Int → a** | left-associative, priority 9 |

The infix version of `nth`. It receives its operands in reverse order, allowing for expressions in the form `ls !! n`.

### 1.2.8 Sorting Operations

Sort lists.

**sort** :: *Orderable a => [a] → [a]*

Sorts a list in ascending order.

### 1.2.9 Zipping Operations

Operations that deal with tuples and lists.

**zip** :: $[a] \to [b] \to [(a, b)]$

Takes two lists and returns a list composed of corresponding pairs. It the lists have different lengths, elements of the larger one are discarded.

**zipWith** :: $(a \to b \to c) \to [a] \to [b] \to [c]$

Takes two lists and a function, returning a list composed of the result of applying the function to corresponding elements in each list. It the lists have different lengths, elements of the larger one are discarded.

**unzip** :: $[(a, b)] \to ([a], [b])$

Takes a list of pairs, returning a pair of lists, each containing the corresponding components of the original list.

## 1.3 String Conversion Operations

Converts values from and to strings.

**parseInt** :: $String \rightarrow Int$

Converts a string into an integer. The only representation accepted is decimal (without a leading +), and the function raises an exception if parsing fails.

**printInt** :: $Int \rightarrow String$

Converts an integer value into a string.

**parseBool** :: $String \rightarrow Bool$

Converts a string into a boolean. Valid strings are `"true"` and `"false"`.

**printBool** :: $Bool \rightarrow String$

Converts a boolean value into a string.