`Int` is fixed width, its exact size depending on the hardware. It silently overflows.

`Integer` is arbitrary sized.

Haskell compilers writers focus their effort on making `Double` efficient.

The combination of `::` and the type after it.

The function name, a space, and then the arguments separated by spaces too.

```
compare 3 2
```

A type constructed from other types, like lists and tuples.

A variable that ranges over types instead of values.

Type variables must start with a lowercase letter, and type names with an uppercase letter.

Both the type name and it's sole value are written `()`.

There are no 1-tuples.

Given a number `n` and a list, `take` returns the first `n` elements of the list, while `drop` returns all *but* the first `n` elements of the list.

`fst` and `snd`

It is the tightest association, and runs left to right.

```
a b c d
```

is equivalent to

```
(((a b) c) d)
```

```
ghci> lines "the quick\nbrown fox\njumps"
["the quick","brown fox","jumps"]
```

Type type of the function's result will begin with `IO`:

```
ghci> :type readFile
readFile :: FilePath -> IO String
```

- `if` expressions result in a value.
- The parens around the condition are inferred.
- There's a `then` keyword to indicate the first branch.

It indicates that a line *continues* an indentation instead of starting a new one.

A function that indicates whether a list is empty.

- Delayed evaluation: evaluating an expression only when the result is required by a calling function.
- Short-circuit evaluation: evaluating an expression only to the extent it is required by the calling function.
- Applicative-order evaluation: evaluating an expression at most once.

`putStrLn` only takes `String` arguments, `print` takes anything instance of `Show`.

In Scala short-circuiting is a result of by-name parameters, a way of achieving the standard Haskell non-strict evaluation. By-name parameters do not adhere to applicative-order evaluation. Haskell doesn't do anything special, expressions always short-circuit.

[1..]

`

last :: [a] -> a
`

"takes a list, all of whose elements have some type `a`, and returns a value of the same type `a`"

One that has type variables in its signature, indicating that some of its arguments can be of any type.

This is an example of parametric polymorphism.

Subtype polymorphism. Haskell isn't object-oriented so it doesn't offer this polymorphism.

Coercion polymorphism. Haskell avoids it in even the simplest of cases. *E.g.*, integers are not converted to floating point numbers.

```
ghci> :t take
take :: Int -> [a] -> [a]

Int -> ([a] -> [a])
```