Variables begin with a lowercase letter.
Type names begin with an uppercase letter.

`--` comment out the rest of the line.

```
{- multiline comment -}
```

(nesting is allowed)

`True` and `False` are of type `Bool`.

The type is strict, only admitting those two literals.

`&&`, `||`, `not`

'==` and `/=`

... strict, since there's no coercion. They have integer and floating point versions.

truncate

```
ord :: Char -> Int
chr :: Int -> Char
```

A list of `Chars`, that is, `[Char]`.

show whatever

Surrounding an identifier with back ticks makes it infix.

Surrounding an operator with parens makes it prefix.

```
ghci> ['a', 'b', 'c'] !! 0
'a'
```

`elem`, `notElem`

nub

concat

```
fst, snd
```

`[a..b]` is a list of all the values from `a` to `b`, inclusive.

`[a..]` is an infinite list from `a` up.

```
[expr | generatorOrGuard1, ... ,generatorOrGuardN]
```

Guards are just expressions that result in `Bool`. No `if` is used.

```
\params -> result
```

```
expression where declarations

let declarations in expressions
```

The expression can have multiple new variables in it.

```
case expr1 of
  expr2 -> ...
  expr3 -> ...
```

```
func params =
   | boolean1 -> ...
   | boolean2 -> ...
```

A data structure whose parts don't exist until they are accessed.

`iterate` takes a function `f` and a starting value `n` and produces a lazy infinite series:

```
(n, f(n), f(f(n)), f(f(f(n))), ...)
```

It comes before a vararg, which is available as a list in the body.

```
(do exprs*)
```

Evaluates the expressions in order and returns the value of the last.

```
(for seq-exprs body-expr)
```

Takes a vector of one or more binding-form/collection-expr pairs, each followed by zero or more modifiers, and yields a lazy sequence of evaluations of expr.

Supported modifiers are: `:let [binding-form expr ...]`, `:while test`, `:when test`.

Use `partial`, followed by a function and fewer than the normal number of arguments.

```
user=> (def equals5 (partial = 5))
#'user/equals5
user=> (equals5 5)
true
```