Basics

Tons of useful functions that get imported by default.

Math

```
type Int
```

An `Int` is a whole number. Valid syntax for integers includes:

```
0
42
9000
0xFF   -- 255 in hexadecimal
0x000A --  10 in hexadecimal
```

Note: `Int` math is well-defined in the range `-2^31` to `2^31 - 1`. Outside of that range, the behavior is determined by the compilation target. When generating JavaScript, the safe range expands to `-2^53` to `2^53 - 1` for some operations, but if we generate WebAssembly some day, we would do the traditional integer overflow. This quirk is necessary to get good performance on quirky compilation targets.

Historical Note: The name `Int` comes from the term integer. It appears that the `int` abbreviation was introduced in ALGOL 68, shortening it from `integer` in ALGOL 60. Today, almost all programming languages use this abbreviation.

```
type Float
```

A `Float` is a floating-point number. Valid syntax for floats includes:

```
0
42
3.14
0.1234
6.022e23   -- == (6.022 * 10^23)
6.022e+23  -- == (6.022 * 10^23)
1.602e-19  -- == (1.602 * 10^-19)
1e3        -- == (1 * 10^3) == 1000
```

Historical Note: The particular details of floats (e.g. `NaN`) are specified by IEEE 754 which is literally hard-coded into almost all CPUs in the world. That means if you think `NaN` is weird, you must successfully overtake Intel and AMD with a chip that is not backwards compatible with any widely-used assembly language.

```
(+) : number -> number -> number
```

Add two numbers. The `number` type variable means this operation can be specialized to `Int -> Int -> Int` or to `Float -> Float -> Float`. So you can do things like this:

```
3002 + 4004 == 7006  -- all ints
3.14 + 3.14 == 6.28  -- all floats
```

You *cannot* add an `Int` and a `Float` directly though. Use functions like toFloat or round to convert both values to the same type. So if you needed to add a list length to a `Float` for some reason, you could say one of these:

```
3.14 + toFloat (List.length [1,2,3]) == 6.14
round 3.14 + List.length [1,2,3]     == 6
```

Note: Languages like Java and JavaScript automatically convert `Int` values to `Float` values when you mix and match. This can make it difficult to be sure exactly what type of number you are dealing with. When you try to *infer* these conversions (as Scala does) it can be even more confusing. Elm has opted for a design that makes all conversions explicit.

`(-) : number -> number -> number`

Subtract numbers like `4 - 3 == 1`.

See `(+)` for docs on the `number` type variable.

`(*) : number -> number -> number`

Multiply numbers like `2 * 3 == 6`.

See `(+)` for docs on the `number` type variable.

`(/) : Float -> Float -> Float`

Floating-point division:

```
10 / 4 == 2.5
11 / 4 == 2.75
12 / 4 == 3
13 / 4 == 3.25
14 / 4 == 3.5

-1 / 4 == -0.25
-5 / 4 == -1.25
```

`(//) : Int -> Int -> Int`

Integer division:

```
10 // 4 == 2
11 // 4 == 2
12 // 4 == 3
```

```
13 // 4 == 3
14 // 4 == 3

-1 // 4 == 0
-5 // 4 == -1
```

Notice that the remainder is discarded, so `3 // 4` is giving output similar to `truncate (3 / 4)`.

It may sometimes be useful to pair this with the `remainderBy` function.

`(^) : number -> number -> number`

Exponentiation

```
3^2 == 9
3^3 == 27
```

Int to Float / Float to Int

`toFloat` `: Int -> Float`

Convert an integer into a float. Useful when mixing `Int` and `Float` values like this:

```
halfOf : Int -> Float
halfOf number =
  toFloat number / 2
```

`round` `: Float -> Int`

Round a number to the nearest integer.

```
round 1.0 == 1
round 1.2 == 1
round 1.5 == 2
round 1.8 == 2

round -1.2 == -1
round -1.5 == -1
round -1.8 == -2
```

`floor` `: Float -> Int`

Floor function, rounding down.

```
floor 1.0 == 1
floor 1.2 == 1
floor 1.5 == 1
floor 1.8 == 1

floor -1.2 == -2
floor -1.5 == -2
floor -1.8 == -2
```

```
ceiling : Float -> Int
```

Ceiling function, rounding up.

```
ceiling 1.0 == 1
ceiling 1.2 == 2
ceiling 1.5 == 2
ceiling 1.8 == 2

ceiling -1.2 == -1
ceiling -1.5 == -1
ceiling -1.8 == -1
```

```
truncate : Float -> Int
```

Truncate a number, rounding towards zero.

```
truncate 1.0 == 1
truncate 1.2 == 1
truncate 1.5 == 1
truncate 1.8 == 1

truncate -1.2 == -1
truncate -1.5 == -1
truncate -1.8 == -1
```

Equality

```
(==) : a -> a -> Bool
```

Check if values are “the same”.

Note: Elm uses structural equality on tuples, records, and user-defined union types. This means the values `(3, 4)` and `(3, 4)` are definitely equal. This is not true in languages like JavaScript that use reference equality on objects.

Note: Do not use `(==)` with functions, JSON values from `elm/json`, or regular expressions from `elm/regex`. It does not work. It will crash if possible. With JSON values, decode to Elm values before doing any equality checks!

Why is it like this? Equality in the Elm sense can be difficult or impossible to compute. Proving that functions are the same is undecidable, and JSON values can come in through ports and have functions, cycles, and new JS data types that interact weirdly with our equality implementation. In a future release, the compiler will detect when `(==)` is used with problematic types and provide a helpful error message at compile time. This will require some pretty serious infrastructure work, so the stopgap is to crash as quickly as possible.

```
(/=) : a -> a -> Bool
```

Check if values are not “the same”.

So `(a /= b)` is the same as `(not (a == b))`.

Comparison

These functions only work on `comparable` types. This includes numbers, characters, strings, lists of comparable things, and tuples of comparable things.

```
(<) : comparable -> comparable -> Bool
```

```
(>) : comparable -> comparable -> Bool
```

```
(<=) : comparable -> comparable -> Bool
```

```
(>=) : comparable -> comparable -> Bool
```

```
max : comparable -> comparable -> comparable
```

Find the larger of two comparables.

```
max 42 12345678 == 12345678
max "abc" "xyz" == "xyz"
```

```
min : comparable -> comparable -> comparable
```

Find the smaller of two comparables.

```
min 42 12345678 == 42
min "abc" "xyz" == "abc"
```

```
compare : comparable -> comparable -> Order
```

Compare any two comparable values. Comparable values include `String`, `Char`, `Int`, `Float`, or a list or tuple containing comparable values. These are also the only values that work as `Dict` keys or `Set` members.

```
compare 3 4 == LT
compare 4 4 == EQ
compare 5 4 == GT
```

```
type Order
    = LT
    | EQ
    | GT
```

Represents the relative ordering of two things. The relations are less than, equal to, and greater than.

Booleans

```
type Bool
    = True
    | False
```

A “Boolean” value. It can either be `True` or `False`.

Note: Programmers coming from JavaScript, Java, etc. tend to reach for boolean values way too often in Elm. Using a union type is often clearer and more reliable. You can learn more about this from Jeremy here or from Richard here.

```
not : Bool -> Bool
```

Negate a boolean value.

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

```
(&&) : Bool -> Bool -> Bool
```

The logical AND operator. `True` if both inputs are `True`.

```
True  && True  == True
True  && False == False
False && True  == False
False && False == False
```

Note: When used in the infix position, like `(left && right)`, the operator short-circuits. This means if `left` is `False` we do not bother evaluating `right` and just return `False` overall.

```
(||) : Bool -> Bool -> Bool
```

The logical OR operator. `True` if one or both inputs are `True`.

```
True  || True  == True
True  || False == True
False || True  == True
False || False == False
```

Note: When used in the infix position, like `(left || right)`, the operator short-circuits. This means if `left` is `True` we do not bother evaluating `right` and just return `True` overall.

```
xor : Bool -> Bool -> Bool
```

The exclusive-or operator. `True` if exactly one input is `True`.

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

Append Strings and Lists

(++) : appendable -> appendable -> appendable

Put two appendable things together. This includes strings and lists.

```
"hello" ++ "world" == "helloworld"
[1,1,2] ++ [3,5,8] == [1,1,2,3,5,8]
```

Fancier Math

modBy : Int -> Int -> Int

Perform modular arithmetic. A common trick is to use (n mod 2) to detect even and odd numbers:

```
modBy 2 0 == 0
modBy 2 1 == 1
modBy 2 2 == 0
modBy 2 3 == 1
```

Our `modBy` function works in the typical mathematical way when you run into negative numbers:

```
List.map (modBy 4) [ -5, -4, -3, -2, -1,  0,  1,  2,  3,
--                 [  3,  0,  1,  2,  3,  0,  1,  2,  3,
```

Use `remainderBy` for a different treatment of negative numbers, or read Daan Leijen's Division and Modulus for Computer Scientists for more information.

remainderBy : Int -> Int -> Int

Get the remainder after division. Here are bunch of examples of dividing by four:

```
List.map (remainderBy 4) [ -5, -4, -3, -2, -1,  0,  1,  2,
--                       [ -1,  0, -3, -2, -1,  0,  1,  2,
```

Use `modBy` for a different treatment of negative numbers, or read Daan Leijen's Division and Modulus for Computer Scientists for more information.

negate : number -> number

Negate a number.

```
negate 42 == -42
negate -42 == 42
negate 0 == 0
```

---

abs : number -> number

Get the absolute value of a number.

```
abs 16   == 16
abs -4   == 4
abs -8.5 == 8.5
abs 3.14 == 3.14
```

---

clamp : number -> number -> number -> number

Clamps a number within a given range. With the expression `clamp 100 200 x` the results are as follows:

```
100     if x < 100
 x      if 100 <= x < 200
200     if 200 <= x
```

---

sqrt : Float -> Float

Take the square root of a number.

```
sqrt  4 == 2
sqrt  9 == 3
sqrt 16 == 4
sqrt 25 == 5
```

---

logBase : Float -> Float -> Float

Calculate the logarithm of a number with a given base.

```
logBase 10 100 == 2
logBase 2 256 == 8
```

---

e : Float

An approximation of e.

Angles

---

degrees : Float -> Float

Convert degrees to standard Elm angles (radians).

```
degrees 180 == 3.141592653589793
```

---

radians : Float -> Float

Convert radians to standard Elm angles (radians).

```
radians pi == 3.141592653589793
```

---

turns : Float -> Float

Convert turns to standard Elm angles (radians). One turn is equal to 360°.

```
turns (1/2) == 3.141592653589793
```

Trigonometry

---

pi : Float

An approximation of pi.

---

cos : Float -> Float

Figure out the cosine given an angle in radians.

```
cos (degrees 60)     == 0.5000000000000001
cos (turns (1/6))    == 0.5000000000000001
cos (radians (pi/3)) == 0.5000000000000001
cos (pi/3)           == 0.5000000000000001
```

---

sin : Float -> Float

Figure out the sine given an angle in radians.

```
sin (degrees 30)     == 0.49999999999999994
sin (turns (1/12))   == 0.49999999999999994
sin (radians (pi/6)) == 0.49999999999999994
sin (pi/6)           == 0.49999999999999994
```

---

tan : Float -> Float

Figure out the tangent given an angle in radians.

```
tan (degrees 45)     == 0.9999999999999999
tan (turns (1/8))    == 0.9999999999999999
tan (radians (pi/4)) == 0.9999999999999999
tan (pi/4)           == 0.9999999999999999
```

acos : Float -> Float

Figure out the arccosine for `adjacent / hypotenuse` in radians:

```
acos (1/2) == 1.0471975511965979 -- 60° or pi/3 radians
```

asin : Float -> Float

Figure out the arcsine for `opposite / hypotenuse` in radians:

```
asin (1/2) == 0.5235987755982989 -- 30° or pi/6 radians
```

atan : Float -> Float

This helps you find the angle (in radians) to an `(x,y)` coordinate, but in a way that is rarely useful in programming. **You probably want `atan2` instead!**

This version takes `y/x` as its argument, so there is no way to know whether the negative signs comes from the `y` or `x` value. So as we go counter-clockwise around the origin from point `(1,1)` to `(1,-1)` to `(-1,-1)` to `(-1,1)` we do not get angles that go in the full circle:

```
atan (  1 /  1 ) ==  0.7853981633974483 --  45° or   pi/4
atan (  1 / -1 ) == -0.7853981633974483 -- 315° or 7*pi/4
atan ( -1 / -1 ) ==  0.7853981633974483 --  45° or   pi/4
atan ( -1 /  1 ) == -0.7853981633974483 -- 315° or 7*pi/4
```

Notice that everything is between `pi/2` and `-pi/2`. That is pretty useless for figuring out angles in any sort of visualization, so again, check out `atan2` instead!

atan2 : Float -> Float -> Float

This helps you find the angle (in radians) to an `(x,y)` coordinate. So rather than saying `atan (y/x)` you say `atan2 y x` and you can get a full range of angles:

```
atan2  1  1 ==  0.7853981633974483 --  45° or   pi/4 radia
atan2  1 -1 ==  2.356194490192345  -- 135° or 3*pi/4 radia
atan2 -1 -1 == -2.356194490192345  -- 225° or 5*pi/4 radia
atan2 -1  1 == -0.7853981633974483 -- 315° or 7*pi/4 radia
```

Polar Coordinates

toPolar : (Float, Float) -> (Float, Float)

Convert Cartesian coordinates (x,y) to polar coordinates (r,θ).

```
toPolar (3, 4) == ( 5, 0.9272952180016122)
toPolar (5, 12) == (13, 1.1760052070951352)
```

```
toPolar (3,12) == (13, 1.176005207095135)
```

fromPolar : (Float, Float) -> (Float, Float)

Convert polar coordinates (r,θ) to Cartesian coordinates (x,y).

```
fromPolar (sqrt 2, degrees 45) == (1, 1)
```

Floating Point Checks

isNaN : Float -> Bool

Determine whether a float is an undefined or unrepresentable number. NaN stands for *not a number* and it is a standardized part of floating point numbers.

```
isNaN (0/0)      == True
isNaN (sqrt -1) == True
isNaN (1/0)      == False  -- infinity is a number
isNaN 1          == False
```

isInfinite : Float -> Bool

Determine whether a float is positive or negative infinity.

```
isInfinite (0/0)      == False
isInfinite (sqrt -1) == False
isInfinite (1/0)      == True
isInfinite 1          == False
```

Notice that NaN is not infinite! For float `n` to be finite implies that `not (isInfinite n || isNaN n)` evaluates to `True`.

Function Helpers

identity : a -> a

Given a value, returns exactly the same value. This is called the identity function.

always : a -> b -> a

Create a function that *always* returns the same value. Useful with functions like `map`:

```
List.map (always 0) [1,2,3,4,5] == [0,0,0,0,0]

-- List.map (\_ -> 0) [1,2,3,4,5] == [0,0,0,0,0]
-- always = (\x _ -> x)
```

`(<|) : (a -> b) -> a -> b`

Saying `f <| x` is exactly the same as `f x`.

It can help you avoid parentheses, which can be nice sometimes. Maybe you want to apply a function to a `case` expression? That sort of thing.

`(|>) : a -> (a -> b) -> b`

Saying `x |> f` is exactly the same as `f x`.

It is called the “pipe” operator because it lets you write “pipelined” code. For example, say we have a `sanitize` function for turning user input into integers:

```
-- BEFORE
sanitize : String -> Maybe Int
sanitize input =
  String.toInt (String.trim input)
```

We can rewrite it like this:

```
-- AFTER
sanitize : String -> Maybe Int
sanitize input =
  input
    |> String.trim
    |> String.toInt
```

Totally equivalent! I recommend trying to rewrite code that uses `x |> f` into code like `f x` until there are no pipes left. That can help you build your intuition.

Note: This can be overused! I think folks find it quite neat, but when you have three or four steps, the code often gets clearer if you break out a top-level helper function. Now the transformation has a name. The arguments are named. It has a type annotation. It is much more self-documenting that way! Testing the logic gets easier too. Nice side benefit!

`(<<) : (b -> c) -> (a -> b) -> a -> c`

Function composition, passing results along in the suggested direction. For example, the following code checks if the square root of a number is odd:

```
not << isEven << sqrt
```

You can think of this operator as equivalent to the following:

```
(g << f)  ==  (\x -> g (f x))
```

So our example expands out to something like this:

```
\n -> not (isEven (sqrt n))
```

`(>>) : (a -> b) -> (b -> c) -> a -> c`

Function composition, passing results along in the suggested direction. For example, the following code checks if the square root of a number is odd:

```
sqrt >> isEven >> not
```

---

type Never

A value that can never happen! For context:

- The boolean type `Bool` has two values: `True` and `False`
- The unit type `()` has one value: `()`
- The never type `Never` has no values!

You may see it in the wild in `Html Never` which means this HTML will never produce any messages. You would need to write an event handler like `onClick ??? : Attribute Never` but how can we fill in the question marks?! So there cannot be any event handlers on that HTML.

You may also see this used with tasks that never fail, like `Task Never ()`.

The `Never` type is useful for restricting *arguments* to a function. Maybe my API can only accept HTML without event handlers, so I require `Html Never` and users can give `Html msg` and everything will go fine. Generally speaking, you do not want `Never` in your return types though.

---

never : Never -> a

A function that can never be called. Seems extremely pointless, but it *can* come in handy. Imagine you have some HTML that should never produce any messages. And say you want to use it in some other HTML that *does* produce messages. You could say:

```
import Html exposing (..)

embedHtml : Html Never -> Html msg
embedHtml staticStuff =
  div []
    [ text "hello"
    , Html.map never staticStuff
    ]
```

So the `never` function is basically telling the type system, make sure no one ever calls me!
