OCAML TYPING!

OVERVIEW

- OCaml is statically typed and VERY strict with types
- certain operators are reserved for certain types
  - → bool: &&, ||
  - → int: +, -, *, /
  - → float: +., -., *., /. (same as int but with a .)
  - → string: ^ (concat)
  - → etc.
- Also certain rules

  if ____ then ____ else ____
  ↑ bool ↑ 'a ↑ 'a

  * the two branches MUST return the same type

  a :: [ ] — 'a list

  a :: b :: [ ] → 'a list — 'a 'a 'a list

  * in ocaml, all elements in a list have the same type

  a :: b :: c — 'a 'a 'a list

  the last thing in a cons operator will be a list, so whatever type that came before that + list
- comparison operators: <, >, =

  * arguments being compared MUST be the same type

EXAMPLES

write an expression that matches the type

(a) float → float → int list

* a function with two parameters, both are floats, and it returns an int list *

```
fun x y -> int_of_float (x +. y) :: []
```

→ conversion functions are helpful to know!

- int_of_float, float_of_int, string_of_int, string_of_float, etc.

(b) float * int → int * float → int

* a function that takes in two tuples, one (float * int) and the other (int * float), and returns an int

```
fun (a,b) (c,d) -> int_of_float (a +. a) + b + c
```

pattern matching directly like this allows us to access individual elements of the tuple

(c) 'a → ('a → 'b) → 'b

* working with generic types, a bit more difficult
* a function with two parameters, one is 'a, the other is a function that takes in a 'a and returns a 'b, and in the end we wanna return a 'b

```
fun x f -> f x
```

* x will be 'a
* f will be 'a -> 'b (we don't know if f returns the same type as x, so we say it returns 'b
* f x is 'b then, because we pass in x for f, and get back a 'b

Given an expression, write the type:

(a) `fun h g z -> (h) ((g 3) +. 4.0)) :: z`

z is at the end of a cons, so it's a list. The element being added is the result of h, so z is a 'a list

takes in the result of ((g 3) +. 4.0), so input is a float, but nothing is present that specifies the type of the output, so output will be generic 'a.

We see +. being used, so (g 3) returns a float, g takes in 3 (an int) so type of g is (int -> float)

Putting it all together:

h: (float -> 'a)

g: (int -> float)

z: 'a list

return: 'a list

(float -> 'a) -> (int -> float) -> 'a list -> 'a list

(b) `fun a b c d -> (a b) < (c d)`

b gets passed into a, we don't know the type, so we'll say generic 'a

d gets passed in to c, we don't know the type of d, or if it's the same as b, so we'll say generic 'c

comparison means (a b) and (c d) have the same type, which means that the functions a and c return the same type. since we don't know exactly what type, we'll say 'b

Putting that all together:

a: ('a -> 'b)

b: 'a

c: ('c -> 'b)

d: 'c

Return: bool

('a -> 'b) -> 'a -> ('c -> 'b) -> 'c -> bool

(c) `fun a b c -> match a with`

matching a with a list pattern, meaning a must be ('a, 'b, 'c) list

```
| [] -> b
| (x,y,z) :: t -> c :: []
```

b has to match the return of the other branch, so we have 'd list

c is an element of a list, idk the element so 'd

Putting it all together:

a: ('a * 'b * 'c) list

b: 'd list

c: 'd

Return: 'd list

('a * 'b * 'c) list -> 'd list -> 'd -> 'd list