Question 1: What is the type of this function?

```ocaml
let cf y x = y;;
```

Out: `val cf : 'a -> 'b -> 'a = <fun>`

Question 2: What does `(cf y)` return?

It returns a constant function.

Question 3: We have the following: `let add a b = a + b;;`

Use a partial application of `add` to define an increment function:

```ocaml
In : let increment = ???
```

```ocaml
In : let increment = add 1;;
```
What is the type of \( f \)?

\[
\text{let } f \ x \ y \ z = x \ z \ (y \ z) \text{;}
\]

Step 1: analyze the right-hand side expression

Function

\[
\begin{align*}
type \ z & : \quad 'a \\
\text{return-type} \ y & : \quad 'b \\
\text{return-type} \ x & : \quad 'c \\
\end{align*}
\]

Step 2: what are the unknown types?

\[
\begin{align*}
\text{input-type} \ y & : \quad 'a \\
\text{input-type} \ x & : \quad 'a \to 'b \\
\end{align*}
\]

Step 3: set those types.

\[
\begin{align*}
type \ y & : \quad 'a \to 'b \\
type \ x & : \quad 'a \to 'b \to 'c \\
type \ z & : \quad 'a \\
\end{align*}
\]

Step 4: infer the input types.

\[
\begin{align*}
\text{let } f \ x \ y \ z = x \ z \ (y \ z) ; ;
\end{align*}
\]

Step 5: infer all types.

\[
\begin{align*}
\text{val } f : ( 'a \to 'b \to 'c ) \to ( 'a \to 'b ) \to 'a \to 'c
\end{align*}
\]

Step 6: infer function type.
Question 4: Is this function tail-recursive? Why?

let rec exists p = function
| [] -> false
| x::xs -> if p x then true else exists p xs

It is...

let rec exists p = function
| [] -> false
| x::xs ->
  if p x then true else
  exists p xs
An example:
perception-action loops (basic building block of autonomy)

\[
\text{while(true)} \\
| \text{get sensor data} \\
| \text{act upon sensor data} \\
| \text{repeat}
\]
Sequential programs - examples include:

- Exhaustive search
  - search a book for a keyword
  - search a graph for the optimal path
- Data processing
  - image processing (enhance / compress)
  - outlier removal / de-noise

Reactive programs - examples include:

- Control tasks
  - flying a plane
  - robot navigation (obstacle avoidance)
- Resource allocation
  - computer processor
  - Mobility-on-Demand (e.g. Uber)
A Pipeline

Producer $\rightarrow$ Filter $\rightarrow$ $\cdots$ $\rightarrow$ Filter $\rightarrow$ Consumer

Produce sequence of items

Filter sequence in stages

Consume results as needed

Lazy lists join the stages together
Lazy Lists — or Streams

Lists of possibly INFINITE length

- elements *computed upon demand*
- *avoids waste* if there are many solutions
- *infinite objects* are a useful abstraction

In OCaml: implement laziness by *delaying evaluation* of the tail

In OCaml: ‘*streams*’ reserved for input/output channels, so we use term ‘*sequences*’
The **type** `unit` has one element: empty tuple `()`

**Uses:**
- Can appear in data-structures (e.g., `unit`-valued dictionary)
- Can be the argument of a function
- Can be the argument or result of a procedure (seen later in course)

Behaves as a tuple, is a constructor, and allowed in pattern matching:

```ocaml
let f () = ... let f = function
    | () ->
```

Expression $E$ not evaluated until the function is applied:

```ocaml
fun () -> E
```

**fun notation enables delayed evaluation!**
Lazy Lists in OCaml

type 'a seq =
  | Nil
  | Cons of 'a * (unit -> 'a seq)

let head (Cons (x, _)) = x
# val head : 'a seq -> 'a = <fun>
Lazy Lists in OCaml

type 'a seq =
| Nil
| Cons of 'a * (unit -> 'a seq)

let head (Cons (x, _)) = x
# val head : 'a seq -> 'a = <fun>

let tail (Cons (_, xf)) = xf ()
# val tail : 'a seq -> 'a seq = <fun>
Lazy Lists in OCaml

```ocaml
type 'a seq =
  | Nil
  | Cons of 'a * (unit -> 'a seq)

let head (Cons (x, _)) = x
# val head : 'a seq -> 'a = <fun>

let tail (Cons (_, xf)) = xf();
# val tail : 'a seq -> 'a seq = <fun>
```

Cons \((x, xf)\) has head \(x\) and tail function \(xf\)

apply \(xf\) to () to evaluate
let rec from k = Cons (k, fun () -> from (k + 1));;
val from : int -> int seq = <fun>

let it = from 1;;
val it : int seq = Cons (1, <fun>)

let it = tail it;;
val it : int seq = Cons (2, <fun>)

tail it;;
- : int seq = Cons (3, <fun>)

```
Recall:
let tail (Cons(_, xf)) = xf ();;
# val tail : 'a seq -> 'a seq
```
Get the first \( n \) elements as a list

\[
\text{let rec get n s =}
\begin{align*}
\text{if n = 0 then } & \text{[]} \\
\text{else } & \\
\text{match } s \text{ with} \\
\text{Nil } & \rightarrow \text{[]} \\
\text{Cons } (x, xf) & \rightarrow x :: \text{get (n - 1) (xf ())}
\end{align*}
\]

\( \text{xf} () \) forces evaluation
get 2 (from 6)
  ⇒ get 2 (Cons (6, fun () -> from (6 + 1)))
  ⇒ 6 :: get 1 (from (6 + 1))
  ⇒ 6 :: get 1 (Cons (7, fun () -> from (7 + 1)))

⇒ 6 :: 7 :: get 0 (from (7 + 1))

⇒ 6 :: 7 :: get 0 (Cons (8, fun () -> from (8 + 1)))
⇒ 6 :: 7 :: []
⇒ [6; 7]
let rec appendq xq yq =
  match xq with
  | Nil -> yq
  | Cons (x, xf) ->
    Cons (x, fun () -> appendq (xf ()) yq)
Joining Two Sequences

let rec appendq xq yq = 
  match xq with 
  | Nil  -> yq 
  | Cons (x, xf)  -> 
    Cons (x, fun () -> appendq (xf ()) yq)

A fair alternative...

let rec interleave xq yq = 
  match xq with 
  | Nil  -> yq 
  | Cons (x, xf)  -> 
    Cons (x, fun () -> interleave yq (xf ()) )
Functionals for Lazy Lists

let rec filter p = function
    | [] -> []
    | x::xs ->
        if p x then
            x :: filter p xs
        else
            filter p xs
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>

We want:
val filterq : ('a -> bool) -> 'a seq -> 'a seq = <fun>
Functionals for Lazy Lists

filtering

let rec filterq p = function
  | Nil -> Nil
  | Cons (x, xf) ->
    if p x then
      Cons (x, fun () -> filterq p (xf ()))
    else
      filterq p (xf ())

let rec iterates f x =
  Cons (x, fun () -> iterates f (f x))

val filterq : ('a -> bool) -> 'a seq -> 'a seq = <fun>
val iterates : ('a -> 'a) -> 'a -> 'a seq = <fun>
Example:

val filterq : ('a -> bool) -> 'a seq -> 'a seq
val iterates : ('a -> 'a) -> 'a -> 'a seq

> let myseq = iterates (fun x -> x + 1) 1;;
# val myseq : int seq = Cons (1, <fun>)

> filterq (fun x -> x = 1) myseq;;
# - : int seq = Cons (1, <fun>)

> filterq (fun x -> x = 100) myseq;;
# - : int seq = Cons (100, <fun>)

> filterq (fun x -> x = 0) myseq;;
......
Reusing Functionals for Lazy Lists

Same Examples, but with no new functions:

```ocaml
c> succ;;
- : int -> int = <fun>
c> succ 1;;
- : 2 = int
c> (=) 1 2
- : bool = false

c> let myseq = iterates succ 1;;
val myseq : int seq = Cons (1, <fun>)
c> filterq ((=) 1) myseq;;
- : int seq = Cons (1, <fun>)
c> filterq ((=) 100) myseq;;
- : int seq = Cons (100, <fun>)
c> filterq ((=) 0) myseq;;
```

Adding 1 has a built-in function!

“=” function, partially applied
Functionals for Lazy Lists

Example:

val filterq : ('a -> bool) -> 'a seq -> 'a seq
val iterates : ('a -> 'a) -> 'a -> 'a seq
val get : int -> 'a seq -> 'a list

> val myseq = iterates (fun x -> x + 1) 1;;
val myseq : int seq Cons (1, <fun>)

> let it = filterq (fun x -> x mod 2 = 0) myseq;;
val it : int seq = Cons (2, <fun>)

> get 5 it;;
- : int list = [2; 4; 6; 8; 10]
Numerical Computations on Infinite Sequences

\[ \text{find } \sqrt{a} \text{ } x_n \]

\[ \text{let } \text{next } a \ x = \left( a \div x + x \right) \div 2.0 \]
Aside: Newton-Raphson Method

Series is:

\[ x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \]
\[ x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \]
\[ x_3 = \vdots \]
\[ x_4 = \vdots \]
\[ x_5 = \vdots \]

So if we want to find \( \sqrt{k} \) we use:

\[ x^2 = k \]
\[ f(x) = x^2 - k \]
\[ f'(x) = 2x \]
Aside: Newton-Raphson Method

Series is:

\[ x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \]
\[ x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \]
\[ x_3 = \vdots \]
\[ x_4 = \vdots \]
\[ x_5 = \vdots \]

So if we want to find \( \sqrt{k} \) we use:

\[ x^2 = k \]
\[ f(x) = x^2 - k \]
\[ f'(x) = 2x \]

\[ x_{n+1} = \frac{1}{2} \left( x_n + \frac{k}{x_n} \right) \]
Numerical Computations on Infinite Sequences

let next a x = (a /. x +. x) /. 2.0

Close enough?

let rec within eps = function
  | Cons (x, xf) -> match xf () with
  | Cons (y, yf) ->
    if abs_float (x -. y) <= eps then y
    else within eps (Cons (y, yf))

\[ x_{n+1} = \frac{1}{2} \left( x_n + \frac{k}{x_n} \right) \]
Numerical Computations on Infinite Sequences

```ocaml
let next a x = (a /. x +. x) /. 2.0

Close enough?

let rec within eps = function
  | Cons (x, xf) ->
    match xf () with
    | Cons (y, yf) ->
      if abs_float (x -. y) <= eps then y
      else within eps (Cons (y, yf))
  | Cons (y, yf) ->
    within eps (Cons (y, yf))

Square Roots!

let root a = within 1e-6 (iterates (next a) 1.0)

> root 3.0;;
- : float = 1.73205080756887719
```