let add_one (x:int) : int = 1 + x
let add_one (x:int) : int = 1 + x

Note: recursive functions with begin with "let rec"
Defining functions

Nonrecursive functions:

```ocaml
let add_one (x:int) : int = 1 + x
let add_two (x:int) : int = add_one (add_one x)
```

Definition of `add_one` must come before use.
Defining functions

Nonrecursive functions:

```ocaml
let add_one (x:int) : int = 1 + x
let add_two (x:int) : int = add_one (add_one x)
```

With a local definition:

```ocaml
let add_two' (x:int) : int = let add_one x = 1 + x in add_one (add_one x)
```

I left off the types. O'Caml figures them out.

Good style: types on top-level definitions.
Types for Functions

Some functions:

```ocaml
let add_one (x:int) : int = 1 + x
let add_two (x:int) : int = add_one (add_one x)
let add (x:int) (y:int) : int = x + y
```

Types for functions:

```ocaml
add_one : int -> int
add_two : int -> int
add : int -> int -> int
```
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:

\begin{align*}
\text{add\_one} : & \text{int} \to \text{int} \\
3 + 4 : & \text{int} \\
\text{add\_one} \ (3 + 4) : & \text{int}
\end{align*}
Multi-argument Functions

Definition:

```ml
let add (x:int) (y:int) : int = x + y
```

Type:

```ml
add : int -> int -> int
```
Multi-argument Functions

Definition:

```haskell
let add (x:int) (y:int) : int = 
    x + y
```

Type:

```haskell
add : int -> int -> int
```

Same as:

```haskell
add : int -> (int -> int)
```
Rule for type-checking functions

General Rule:
If a function $f : T_1 \rightarrow T_2$ and an argument $e : T_1$ then $f e : T_2$

Example:
add : int -> int -> int
3 + 4 : int
add (3 + 4) : ???
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:

\[
\text{add : int} \rightarrow (\text{int} \rightarrow \text{int})
\]

\[
3 + 4 : \text{int}
\]

\[
\text{add} (3 + 4) :
\]
Rule for type-checking functions

General Rule:
If a function \( f : T_1 \to T_2 \) and an argument \( e : T_1 \) then \( f \; e : T_2 \)

Example:
add : int \to (\text{int} \to \text{int})
3 + 4 : \text{int}
add (3 + 4) : \text{int} \to \text{int}
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:

\[
\begin{align*}
\text{add} : & \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 &: \text{int} \\
\text{add} \ (3 + 4) &: \text{int} \rightarrow \text{int} \\
(\text{add} \ (3 + 4)) \ 7 &: \text{int}
\end{align*}
\]
Rule for type-checking functions

General Rule:

If a function $f : T_1 \rightarrow T_2$ and an argument $e : T_1$ then $f \ e : T_2$

Example:

```
add : int \rightarrow\ int \rightarrow\ int
3 + 4 : int
add (3 + 4) : int \rightarrow\ int
add (3 + 4) 7 : int
```

A -> B -> C
same as:
A -> (B -> C)

extra parens
not necessary
One key thing to remember

• If you have a function \( f \) with a type like this:

\[
A \to B \to C \to D \to E \to F
\]

• Then each time you add an argument, you can get the type of the result by knocking off the first type in the series

\[
f\ a1 : B \to C \to D \to E \to F \quad (\text{if } a1 : A)
\]

\[
f\ a1\ a2 : C \to D \to E \to F \quad (\text{if } a2 : B)
\]

\[
f\ a1\ a2\ a3 : D \to E \to F \quad (\text{if } a3 : C)
\]

\[
f\ a1\ a2\ a3\ a4\ a5 : F \quad (\text{if } a4 : D \text{ and } a5 : E)
\]
DEBUGGING TYPE ERRORS
Debugging Type Errors

Type errors can be confusing sometimes. Consider:

```ocaml
let rec concatn s n =
  if n <= 0 then ...
  else
    s ^ (concatn s (n-1))
```
Debugging Type Errors

Type errors can be confusing sometimes. Consider:

```ocaml
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocamlbuild says:

```
Error: This expression has type int but an expression was expected of type string
```
Type Checking Rules

Type errors can be confusing sometimes. Consider:

```ocaml
let rec concatn s n =
  if n <= 0 then ...
  else
    s ^ (concatn s (n-1))
```

ocamlbuild says:

```
Error: This expression has type int but an expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an expression was expected of type int
```
Type errors can be confusing sometimes. Consider:

```ocaml
let rec concatn s n =
  if n <= 0 then ...
  else
    s ^ (concatn s (n-1))
```

`ocamlbuild` says:

```
Error: This expression has type int but an expression was expected of type string
```

`merlin` inside `emacs` points to the error above and gives a second error:

```
Error: This expression has type string but an expression was expected of type int
```
Type Checking Rules

Type errors can be confusing sometimes. Consider:

```
let rec concatn s n =
    if n <= 0 then
        0
    else
        s ^ (concatn s (n-1))
```

they don't agree!

ocamlbuild says:

```
Error: This expression has type int but an expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an expression was expected of type int
```
Type Checking Rules

Type errors can be confusing sometimes. Consider:

```ocaml
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

The type checker points to some place where there is disagreement.

Moral: Sometimes you need to look in an earlier branch for the error even though the type checker points to a later branch. The type checker doesn't know what the user wants.
A Tactic: Add Typing Annotations

```ml
let rec concatn (s:string) (n:int) : string =
    if n <= 0 then
        0
    else
        s ^ (concatn s (n-1))
```

Error: This expression has type int but an expression was expected of type string
Exercise

Given the following code:

```ocaml
let munge b x =
  if not b then
    string_of_int x
  else
    "hello"

let y = 17

What are the types of the following expressions?
(And what must the types of f and g be?)

munge : ??
munge (y > 17) : ??
munge true (f (munge false 3)) : ??
munge true (g munge) : ??
```