Programming Languages and Techniques (CIS120)

Lecture 8

Generics & First-class functions Chapters 8 and 9

Generic Functions and Data

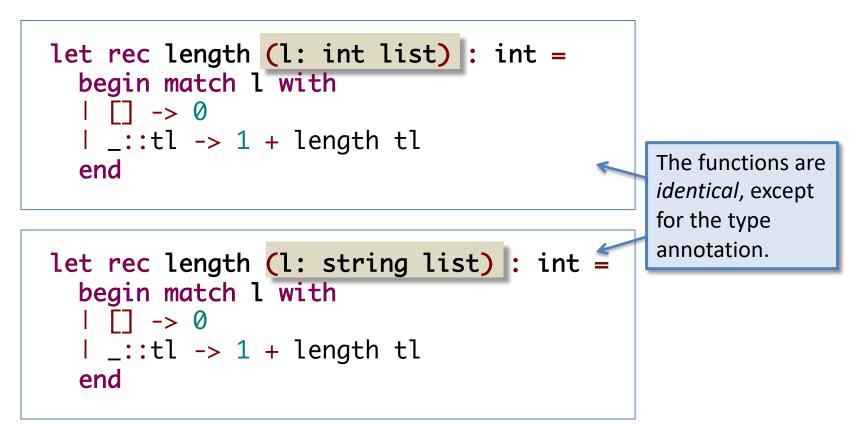
Wow, implementing BSTs took quite a bit of typing... Do we have to do it all again if we want to use BSTs containing strings, and again for characters, and again for floats, and...?

or

How not to repeat yourself, Part I.

Structurally Identical Functions

- Observe: many functions on lists, trees, and other datatypes don't depend on the contents, only on the structure.
- Compare:



Notation for Generic Types

• OCaml allows defining functions with *generic* types

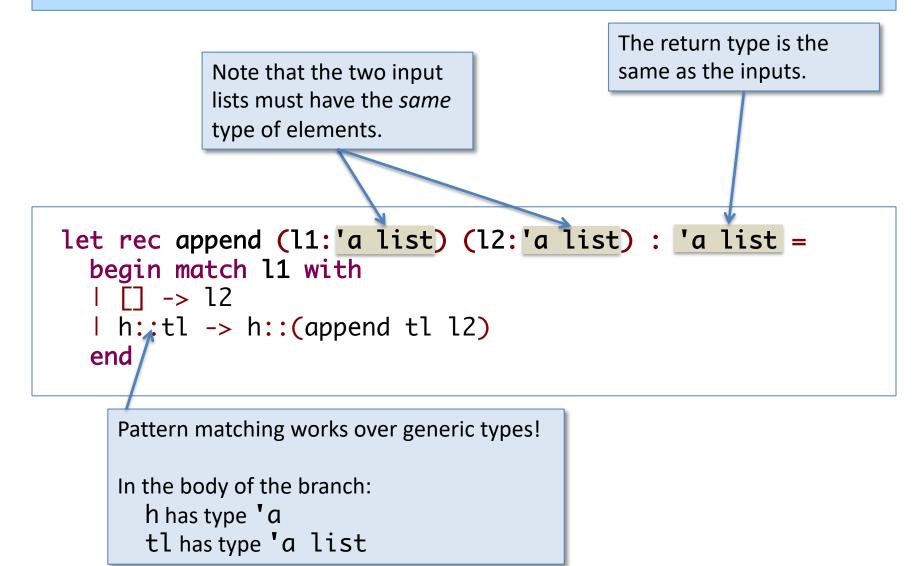
```
let rec length (l:'a list) : int =
    begin match l with
    [] -> 0
    I _::tl -> 1 + (length tl)
    end
```

- Notation: 'a is a type variable, indicating that the function length can be used on a t list for any type t.
- Examples:
 - length [1;2;3] use length on an int list
 - length ["a";"b";"c"]

use length on a string list

• Idea: OCaml fills in 'a whenever length is used

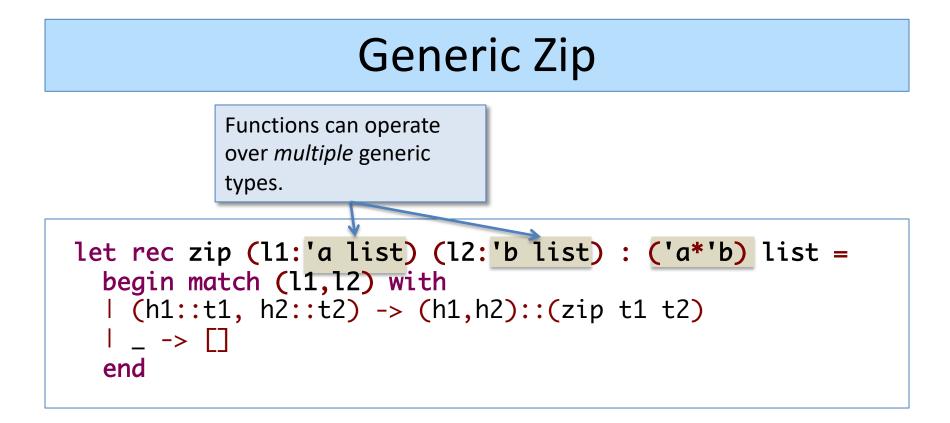
Generic List Append



Zip function

- Combine two lists into one list
 - ignore elements from longer list if they are not the same length

• Does it matter what type of lists these are?



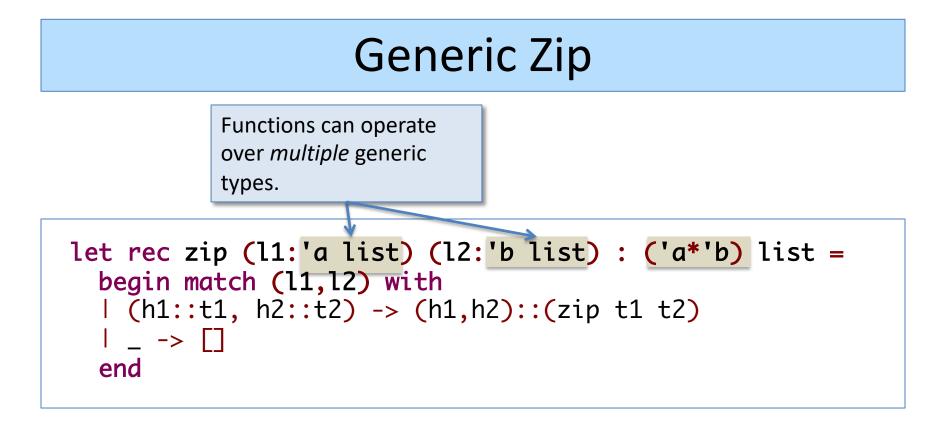
• Distinct type variables *can* be instantiated differently:

zip [1;2;3] ["a";"b";"c"]

- Here, 'a is instantiated to int, 'b to string
- Result is

[(1,"a");(2,"b");(3,"c")]
of type (int * string) list

Intuition: OCaml tracks instantiations of type variables ('a and 'b) and makes sure they are used consistently



• Distinct type variables *do not need to be* instantiated differently:

zip [1;2;3] [4;5;6]

- Here, 'a is instantiated to int, 'b to int
- Result is

[(1,4);(2,5);(3,6)]
of type (int * int) list

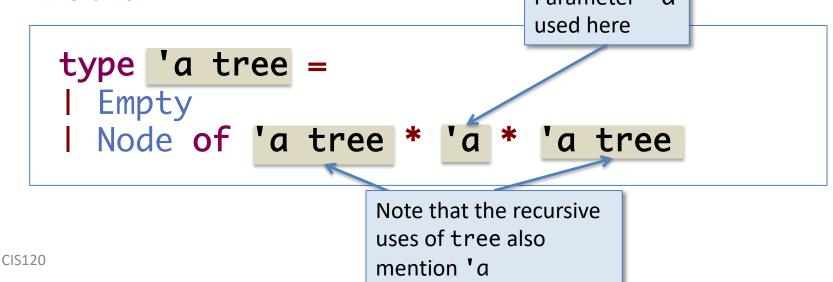
Intuition: OCaml tracks instantiations of type variables ('a and 'b) and makes sure they are used consistently

User-Defined Generic Datatypes

• Recall our integer tree type:

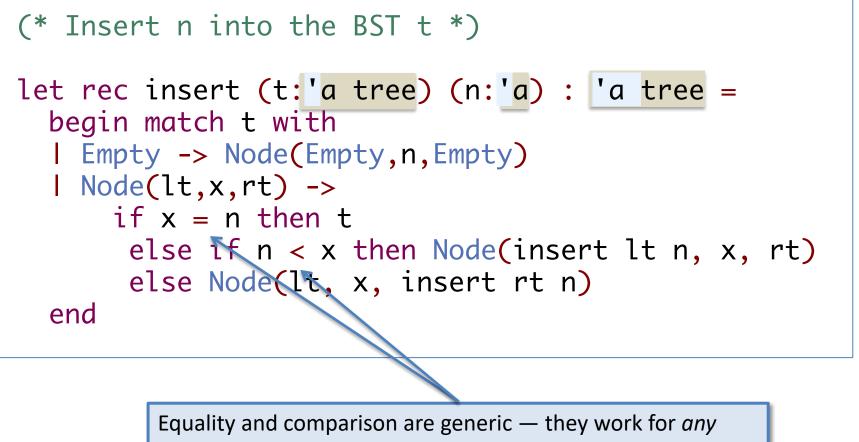
```
type tree =
   Empty
   Node of tree * int * tree
```

 We can define a generic version by adding a type parameter, like this:

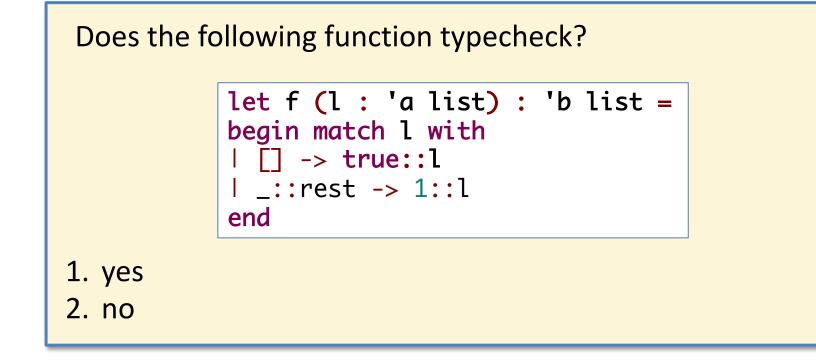


User-Defined Generic Datatypes

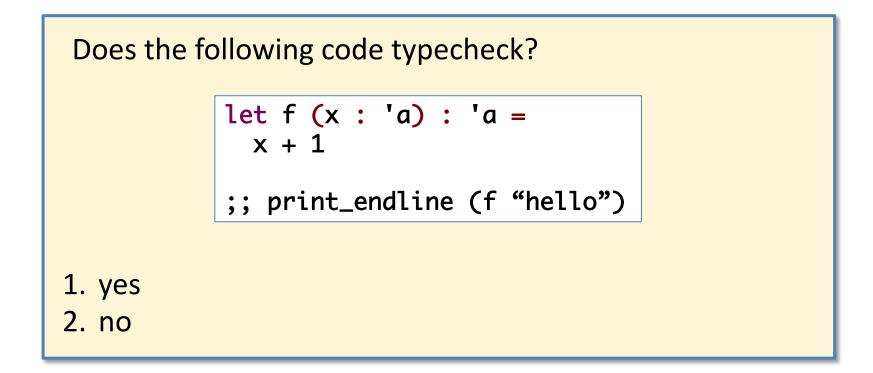
BST operations can be generic too; the only change is to the type annotation



type of data.



Answer: no: even though the return type is generic, the two branches must agree (so that 'b can be consistently instantiated).



Answer: no, the type annotations and uses of f aren't consistent.

However it is a bit subtle: without the use (f "hello") the code *would* be correct – so long as all uses of f provide only 'int' the code is consistent! Despite the "generic" type annotation, f really has type int -> int.

First-class Functions

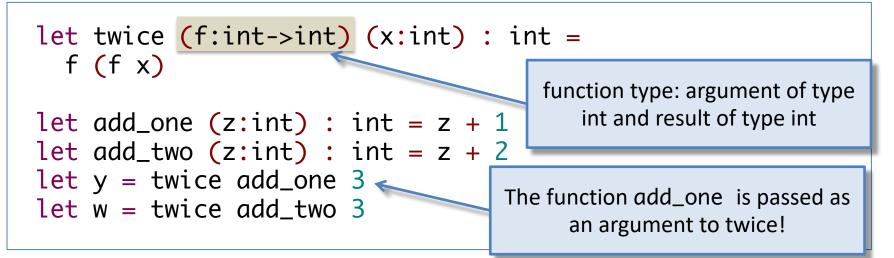
Higher-order Programs

or

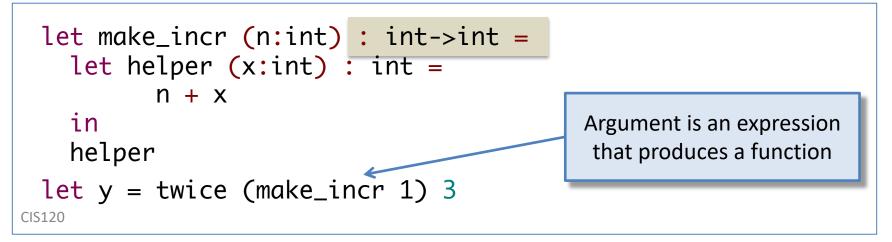
How not to repeat yourself, Part II.

First-class Functions

• You can pass a function as an *argument* to another function:

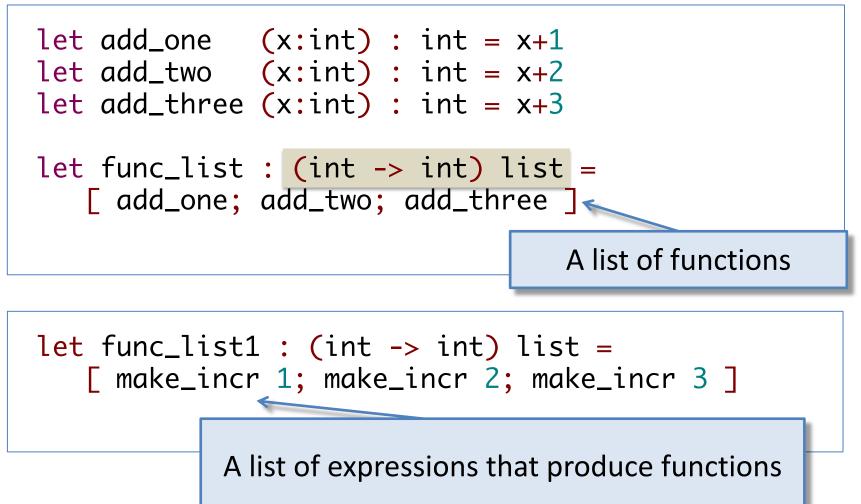


• You can *return* a function as the result of another function.



Functions as Data

• You can store functions in data structures



Simplifying First-Class Functions

```
let twice (f:int->int) (x:int) : int =
  f (f x)
let add_one (z:int) : int = z + 1
```

```
twice add_one 3

\mapsto add_one (add_one 3)

\mapsto add_one (3 + 1)

\mapsto add_one 4

\mapsto 4 + 1

\mapsto 5
```

substitute add_one for f, 3 for x substitute 3 for z in add_one $3+1 \Rightarrow 4$ substitute 4 for z in add_one $4+1 \Rightarrow 5$

Simplifying First-Class Functions

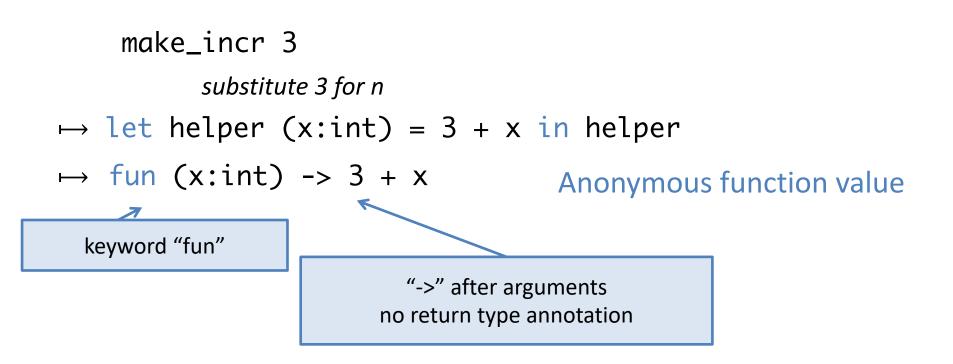
```
let make_incr (n:int) : int->int =
   let helper (x:int) : int = n + x in
   helper
```

make_incr 3
 substitute 3 for n

 Het helper (x:int) = 3 + x in helper
 Helper ???

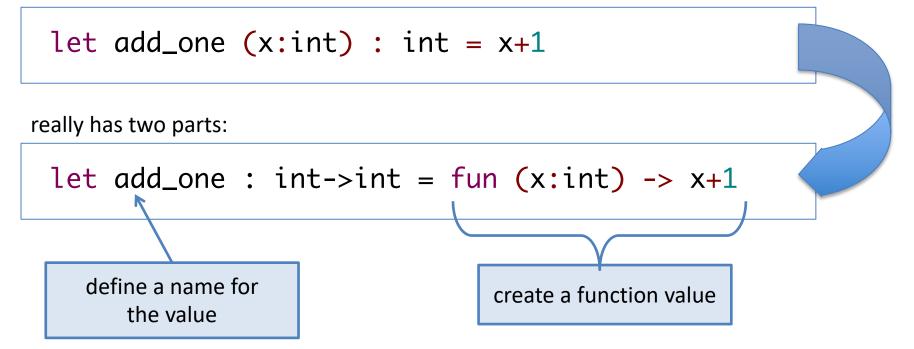
Simplifying First-Class Functions

```
let make_incr (n:int) : int->int =
   let helper (x:int) : int = n + x in
   helper
```



Named function values

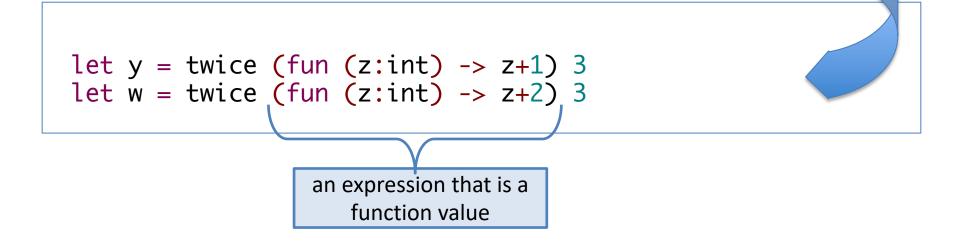
A standard function definition...



The two definitions have the same type and behave exactly the same. (The first is actually just an abbreviation for the second.)

Anonymous functions

```
let add_one (z:int) : int = z + 1
let add_two (z:int) : int = z + 2
let y = twice add_one 3
let w = twice add_two 3
```



Function Types

• Functions have types that look like this:

$$t_{in} \rightarrow t_{out}$$

• Examples:

int -> int
int -> bool * int
int -> int -> int int input
(int -> int) -> int function input

Function Types

• Functions have types that look like this:

 $t_{in} \rightarrow t_{out}$

• Examples:

Parentheses matter!
int -> int
int -> (bool * int)
int -> (int -> int)
int -> (int -> int)
int -> int)
function input

Function Types

Hang on... did we just say that

and

mean the same thing??

Yes!

2 = 1 + 1

The type of a function that takes *two* arguments

int -> int -> int

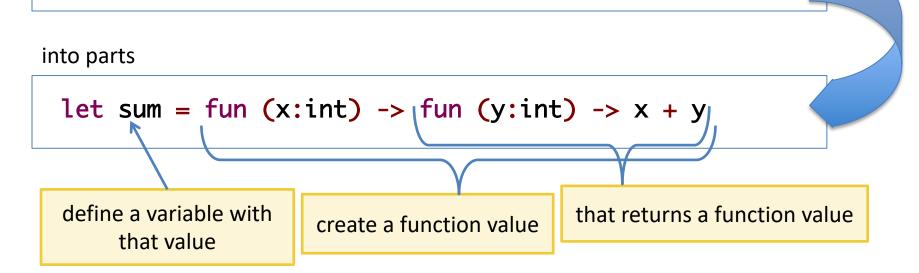
is the same as the type of a function that takes one argument and returns a function that takes one argument.

We can exploit this in OCaml!

Multiple Arguments

We can decompose a standard function definition

```
let sum (x : int) (y:int) : int = x + y
```



The two definitions have the same type and behave exactly the same

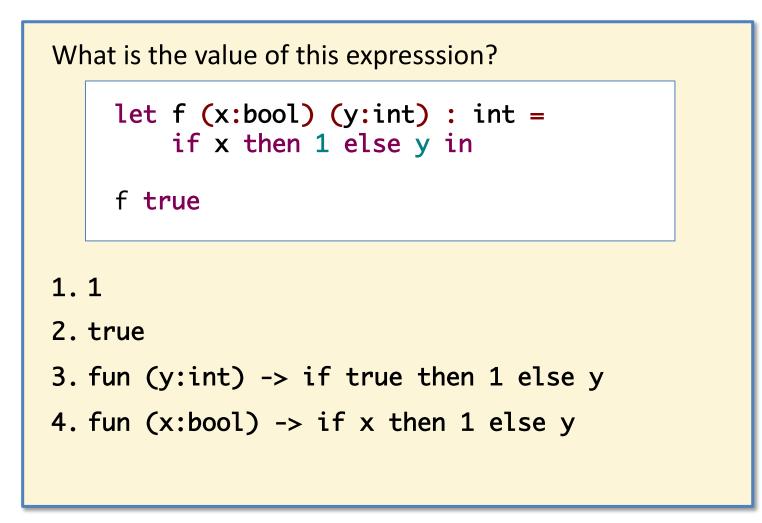
let sum : int -> int -> int

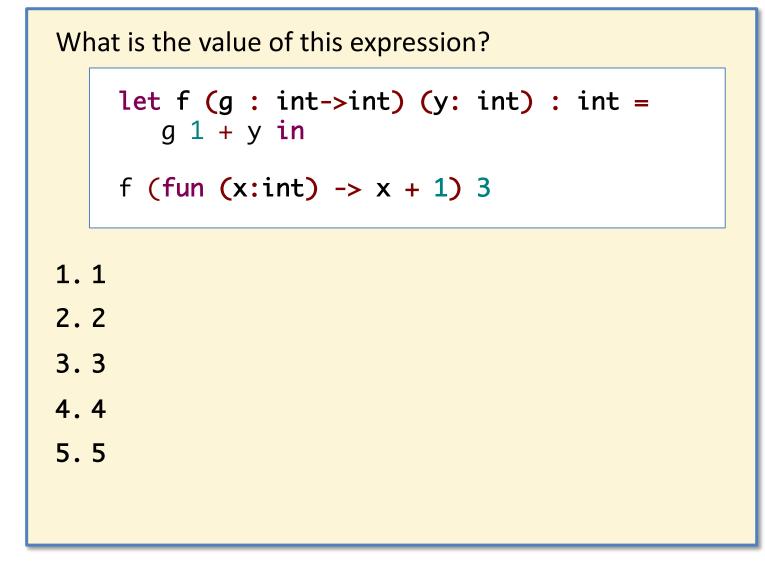
Partial Application

let sum (x : int) (y:int) : int = x + y

sum 3

 $\mapsto (fun (x:int) \rightarrow fun (y:int) \rightarrow x + y) 3 \quad definition$ $\mapsto fun (y:int) \rightarrow 3 + y \qquad substitute 3 for x$





What is the type of this expression?

```
let f (g : int->int) (y: int) : int =
    g 1 + y in
```

```
f (fun (x:int) -> x + 1)
```

```
1. int
2. int -> int
3. int -> int -> int
4. (int -> int) -> int -> int
5. ill-typed
```

What is the type of this expresssion?

```
[ (fun (x:int) -> x + 1);
  (fun (x:int) -> x - 1) ]
```

1. int

```
2. int -> int
```

3. (int -> int) list

4. int list -> int list

5. ill typed