Q: What is the value of \( \text{res} \)?

```plaintext
let f g = 
  let x = 0 in 
  g 2  
let x   = 100 
let h y = x + y 
let res = f h 
```

(a) 0  (b) 2  (c) 100  (d) 102  (e) 12

---

**Static/Lexical Scoping**

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment

- **Static/Lexical:** Determined from the program text
  - Without executing the program

- Very useful for **readability, debugging:**
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence

---

**Immutability: The Colbert Principle**

“A function behaves the same way on Wednesday, as it behaved on Monday, no matter what happened on Tuesday!”
Polymorphism

Polymorphism enables Reuse

- Can reuse generic functions:
  - `map : 'a * 'b -> 'b * 'a`
  - `filter : ('a -> bool) -> 'a list -> 'a list`
  - `rev : 'a list -> 'a list`
  - `length : 'a list -> int`
  - `swap : 'a * 'b -> 'b * 'a`
  - `sort : ('a -> 'a -> bool) -> 'a list -> 'a list`
  - `fold : ...`

- If function (algorithm) is independent of type, can reuse code for all types!

Polymorphic Data Types

- Data types are also polymorphic!

  ```
  type 'a list =
  | Nil
  | Cons of ('a * 'a list)
  ```

- Type is instantiated for each use:

  ```
  Cons(1,Cons(2,Nil)) :
  Cons("a",Cons("b",Nil)) :
  Cons((1,2),Cons((3,4),Nil)) :
  Nil :
  ```

News

- Midterm on Thursday
  - Double-sided “cheat sheet”
  - Printed, if you like

- PA4 due NEXT Friday @ 5p
  - First half relevant for Midterm
Polymorphic Data Types

- Data types are also polymorphic!
  - `type 'a list = Nil | Cons of ('a * 'a list)`
- Type is instantiated for each use:
  - `Cons(1,Cons(2,Nil)) : int list`
  - `Cons("a",Cons("b",Nil)) : string list`
  - `Cons((1,2),Cons((3,4),Nil)) : (int*int) list`
  - `Nil : 'a list`

Datatypes with many type variables

- Multiple type variables
  - `type ('a, 'b) tree = Leaf | Node of 'a* 'b * ('a,'b) tree * ('a,'b) tree`
- Type is instantiated for each use:
  - `Node("alice", 2, Leaf, Leaf)`
  - `Node("charlie", 3, Leaf, Leaf)`
  - `Node("bob", 13, , Node("alice", 2, Leaf, Leaf), Node("charlie", 3, Leaf, Leaf))`

Q: What is the type of `res`?

(a) `(int, string) tree`
(b) `(a, b) tree`
(c) `int tree`
(d) type error
(e) `(string, int) tree`
Q: What is the type of \texttt{res}?

(a) (int, string) tree
(b) ('a,'b) tree
(c) int tree
(d) type error
(e) (string, int) tree

Datatypes with many type variables

- Multiple type variables

```
type ('a,'b) tree = Leaf
| Node of 'a* 'b * ('a,'b) tree * ('a,'b) tree
```

```
let res = Node("bob",13,Node(3, "alice",Leaf, Leaf),Leaf)
```

- Type is instantiated for each use:

```
Node("alice", 2, Leaf, Leaf)
Node("charlie", 3, Leaf, Leaf)
Node("bob", 13,
    , Node("alice", 2, Leaf, Leaf)
    , Node(3, "charlie", Leaf, Leaf))
```

A tricky question: consider this type

```
type ('a, 'b) weirdlist = Nil
| Cons 'a* ('b, 'a) weirdlist
```

Which is a valid Ocaml Expression?

(a) Cons(1, Cons("a", Cons(3.14, Nil)))
(b) Cons(1, Cons("a", Cons(1, Nil)))
(c) Cons(1, Cons("a", Cons("a", Nil)))
(d) Cons(1, Cons(1, Cons("a", Nil)))
(e) Cons(1, Cons(1, Cons(1, Nil)))

Polymorphic Data Structures

- Container data structures independent of type!
- Appropriate type is instantiated at each use:

```
'a list
('a , 'b) tree
('a , 'b) hashtbl ...
```

- Static type checking catches errors early
- Cannot add int key to string hashtable
- Generics: in Java,C#,VB (borrowed from ML)
Type Inference

How DOES Ocaml figure out all the types?!

Polymorphic Types

- Polymorphic types are tricky
- Not always obvious from staring at code
- How to ensure correctness?
- Types (almost) never entered w/ program!

Polymorphic Type Inference

- Computing the types of all expressions
  - At compile time: statically Typed
- Each binding is processed in order
  - Types are computed for each binding
  - For expression and variable bound to
    - Types used for subsequent bindings
- Unlike values (determined at run-time)

Polymorphic Type Inference

- Every expression accepted by ML must have a valid inferred type
- Can have no idea what a function does, but still know its exact type
- A function may never (or sometimes terminate), but will still have a valid type
Example 1

```ocaml
let x = 2 + 3;;
let y = string_of_int x;;
```

Example 2

```ocaml
let x = 2 + 3;;
let y = string_of_int x;;
let inc y = x + y;;
```

Example 4

```ocaml
let rec cat xs =
  match xs with
  | [] -> cat []
  | x::xs -> x^(cat xs)
```

```
let foo x = 
  let (y, z) = x in 
  z-y
```

What's the type of `foo`?

(a) `int`
(b) `int * int`
(c) `int * int -> int`
(d) `int -> int -> int`
(e) Error

```
let rec cat xs =
  match xs with
  | [] -> cat []
  | x::xs -> x^(cat xs)
```

(a) `string -> string`
(b) `string`
(c) `string list -> string list`
(d) `string list -> string`
(e) Error
```ml
Example 5

let rec cat xs =
  match xs with
  | []    -> ""
  | x::xs -> x^(cat xs)

Example 5

let rec map f xs =
  match xs with
  | []     -> []
  | x::xs' ->(f x)::(map f xs')

ML doesn’t know what function does, or even that it finishes only its type!

Example 5

let rec cat xs =
  match xs with
  | []    -> ""
  | x::xs -> x^(cat xs)

Example 5

let rec cat xs =
  match xs with
  | []    -> cat []
  | x::xs -> x^(cat xs)

Example 5

"Generalize" Unconstrained Vars

('a->'b) -> 'a list -> 'b list
```
What is the type of \((\triangleleft\triangleright)\)

\[
\text{let } (\triangleleft\triangleright) \ f \ g \ x = g \ (f \ x) \n\]

(a) \(\text{‘a -> ‘b -> ‘c -> ‘d}\)
(b) \('(\text{‘a->‘b)}->(‘a ->’b)->(‘a ->’b)\)
(c) \((\text{int->char})->(\text{char->bool})->(\text{int->bool})\)
(d) \((\text{int->int})->(\text{int->int})->(\text{int->int})\)
(e) \('(\text{‘a->’b)}->(‘b ->’c)->(‘a ->’c)\)

Example 6

\[
\text{let compose } \ f \ g \ x = f \ (g \ x) \n\]

Example 7

\[
\text{let rec fold } f \ cur \ xs = \\
\text{match } xs \ \text{with} \\
\quad \quad [\quad] -> \ cur \\
\quad \quad | \ x::xs' -> \text{fold } f \ (f \ cur \ x) \ xs' \n\]
(In Class Exercise A)

```
let rec split xs =
  match xs with
  | []    -> ([], [])
  | [x]   -> ([x], [])
  | y::z::xs' ->
    let ys, zs = split xs' in
    (y::ys, z::zs)
```

(In Class Exercise B)

```
let rec merge xs ys =
  match (xs, ys) with
  | ([], _) -> ys
  | (_, []) -> xs
  | (x::xs', y::ys') when x <= y
    -> x :: (merge xs' ys)
  | (x::xs', y::ys')
    -> y :: (merge xs ys')
```

(In Class Exercise C)

```
let rec msort xs =
  match xs with
  | []     -> []
  | x::xs' ->
    let ys, zs = split xs in
    merge (msort ys) (msort zs)
```

Example 11

```
let foo1 f g x =
  if f x
  then x
  else g x
```

Example 11

```
let fool f g x =
  if f x
  then x
  else g x
```
**Example 12**

```ocaml
let foo2 f g x =  
  if f x  
  then x  
  else foo2 f g (g x)
```

**Binary Search Trees**

```
type ('a, 'b) tree =  
  Leaf  
  | Node of 'a * 'b * ('a, 'b) tree * ('a, 'b) tree
```

**Node (key, value, left, right)**

**BST Property:**

`keys in left < key < keys in right`

**Exercise!**

Write a function to lookup keys...

```
type ('a, 'b) tree =  
  Leaf  
  | Node of 'a * 'b * ('a, 'b) tree * ('a, 'b) tree

val lookup : 'a -> ('a, 'b) tree -> 'b option
```

**Node:**

```
Node("bob", 13)  
  , Node("alice", 2, Leaf, Leaf)  
  , Node("charlie", 3, Leaf, Leaf)
```