- Installing OCaml } see CS162 github (section / sec 01)
- Running OCaml }
- OCaml Review
    + Mutation vs. Binding
    + Products
    + Options
- Recursion for CS162
- Program Syntax

Mutation vs Binding

let $x = e_1$ in e_2

Consider:

<Python>

```
x = 1
x = x+1
print(x)
```

⇒ 2

<OCaml>

```
let x = 1 in
let x = x+1 in
print_int(x)
```

⇒ 2

<OCaml>

```
let x=1 in
let y = (let x=x+1 in x) in
print_int(x)
```

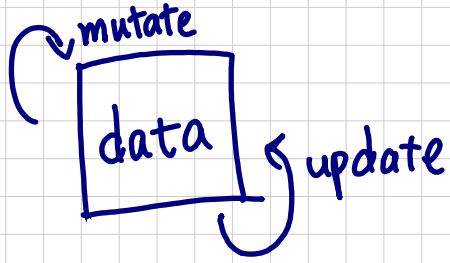
⇒ 1

In OCaml, "variables" don't vary.

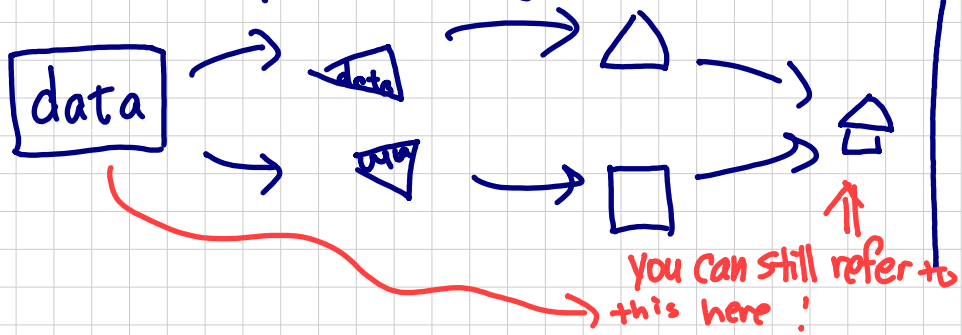
They aren't memory boxes.

Instead, they're names for values that are immutable.

Imperative programming



Functional programming



(Before talking about products / options...)

Everything in OCaml is an expression.

2 kinds of behaviors

Types

What do you think/predict will happen if you evaluate it?

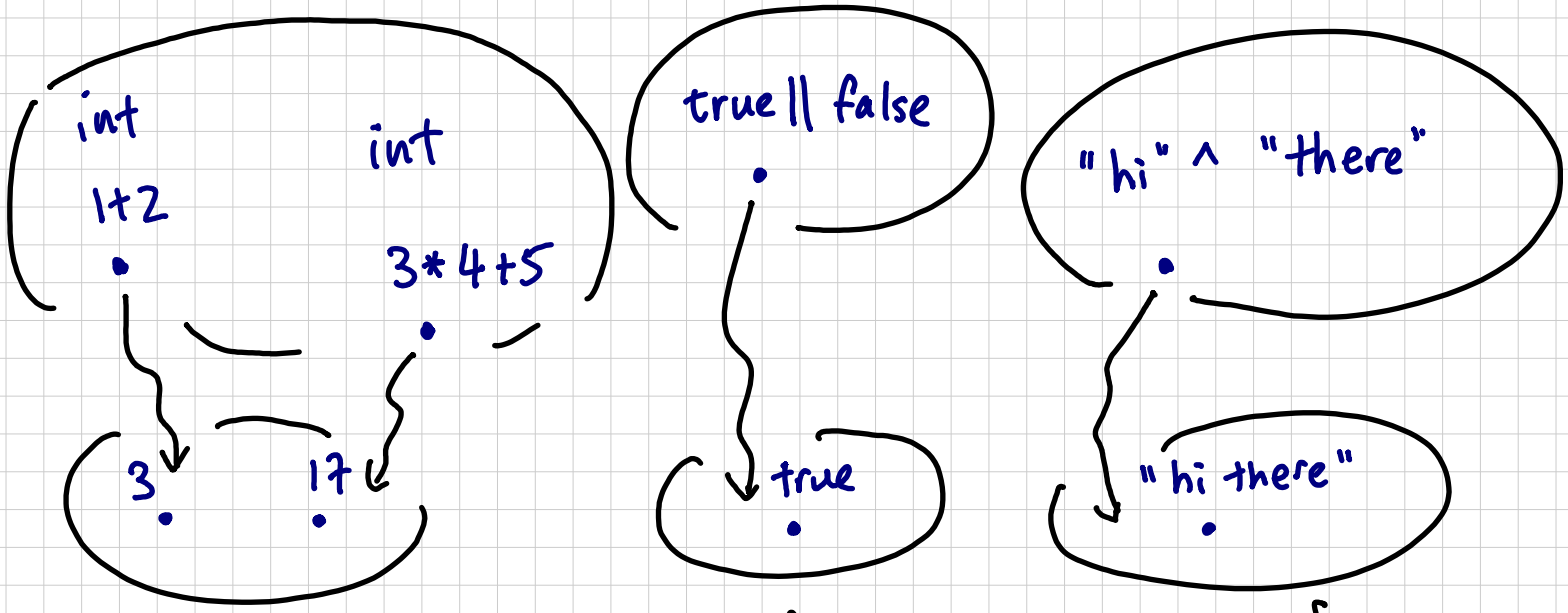
Run-time

What happens if you evaluate it?

Example

Exps

Values



if e: int, you know e must evaluate to something in {0, 1, -1, 2, -2, ...}

can't possibly evaluate to true / false / "hi" / ...

Every type is characterized by 2 kinds of operations:

- Constructors : How do I make something of type t?
- Destructors : How do I use something of type t?

(In OCaml, types are defined constructively.)

Destructors = pattern match

Examples

Type	How to make	How to use
bool	true, false	if b then ... else ... ⟺ match b with | true → ... | false → ...
$t_1 * t_2$	if e_1 has type t_1 e_2 has type t_2 then (e_1, e_2) has type $t_1 * t_2$	if e has type $t_1 * t_2$, then (fst e) extracts 1st component. (snd e) extracts 2nd component ⟺ match e with | (x, y) → ...

Options

Say f is some computation.

- f may succeed with a return value.
- f may fail with no value at all

In OCaml, `type int_option = Some of int | None`

Generic: `type 'a option = Some of 'a | None`

↗ "template" for producing specific option type e.g.
int option, bool option, string option, (int option) option

How to make an int_option?

- If you have int x, Some x.
- If you have nothing, None.

How to use an int_option?

We don't know for sure whether it's Some or None.

⇒ Need to handle both cases.

```
match x with
| Some n → ...
| None → ...
```

Exercises: Do the following functions exist?

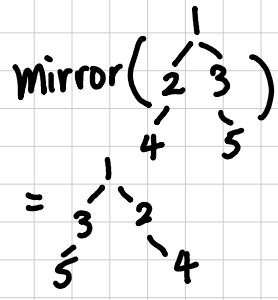
1. int → int option
2. int option → int
3. 'a → 'a option
4. int → 'a optio
5. 'a option → '

Recipes for recursion (for CS162)

1. Identify the structure of the input. (FREE)
2. Decompose the structure (FREE)
3. Handle the base case. (EASY)
4. Call f on the recursive sub-structure. (FREE)
5. Assume the results of 4 is correct. (FREE)
6. Compose the results into an overall sol'n.

↑ Usually the only part that requires insight.

Examples



1. Structure binary tree

```
type tree = Leaf of int | Node of int * tree * tree
```

2. Either Leaf or Node with 2 subtrees.
3. Handle Leaf case
4. Call f on 2 substrees
5. Assume step 4 solves the problem for l & r. Now, construct sol'n to the overall tree.

```
let rec mirror t =
  match t with
  | Leaf(n) -> Leaf(n)
  | Node(n, l, r) ->
    let l' = mirror l in
    let r' = mirror r' in
    Node(n, r', l').
```

mirror(2 / 4) = 2 \ 4 (inductive hypothesis)

mirror(3 \ 5) = 3 / 5

Program Syntax

Python

```
def fib(n):
    if n=0:
        return 0
    elif n=1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

What the editor sees:

"d e f ␣ ␣ f i b (n) : \n ␣ ␣ ␣ ␣ i f ..."

↑ a string of characters

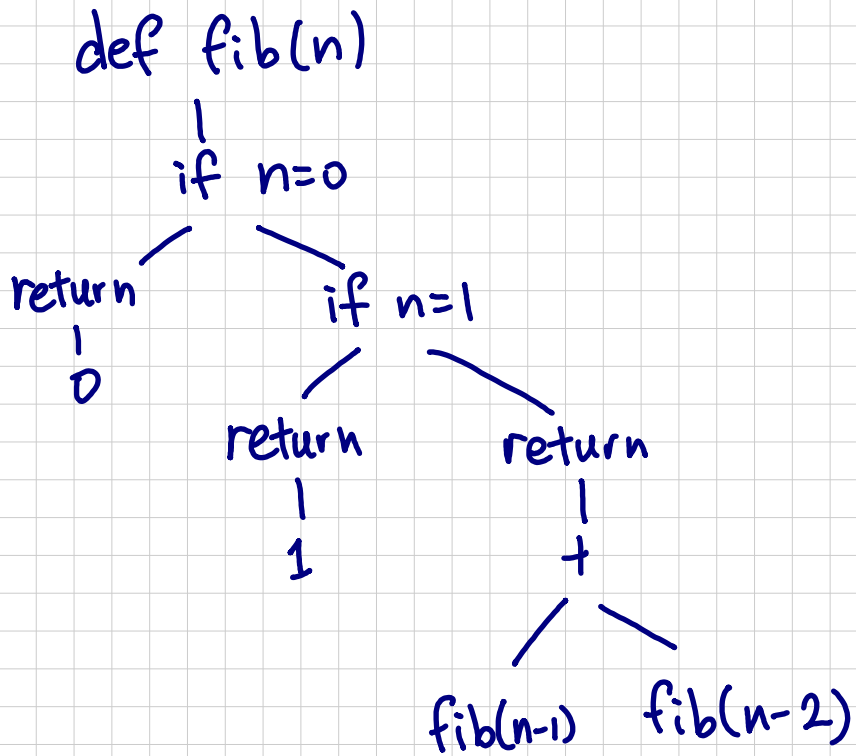
Problem

Imagine you're ...

- the compiler: I want to optimize the recursive calls.
- the interpreter: I want to evaluate fib(10).

Strings have too little structure!

Solution



Use trees!

AKA abstract syntax trees (ASTs)

Contrast with concrete syntax (strings)

- Parser turns strings into ASTs.
- We will give you a parser. You just need to manipulate trees, not strings.
- Take CS160 if you want to know how parsers work.