Section 2

- Review data types in OCaml
- Abstract syntax trees
- Recursion recipe for CS162

Data types in OCaml

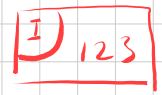
only 2 kinds ⇒

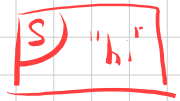
only need to know 2 things

⇓

	Product type	Enum type
	Example: `int * string`	Example: `type t = I of int | S of string`
	Logically: AND	Logically: XOR
How to build?	Q1 `(a, b)`	Q3 `I 123` or `S "hi"`

How to use?	Q2 `let (x,y) = p in ...`
	 `match p with`
	 `| (x,y) → ... x ... y ...`
		Q4 `match x with`
		 `| I x → ... x ...`
		 `| S x → ... x ...`





Useful predefined data types

1) 'a option = None | Some of 'a

2) 'a list = [] | (::) of 'a * 'a list

(Nil) (Cons)

Generic types Instead repeating the same type def over & over, define one template:

```
type int_option
  = Some of int | None
type bool_option
  = Some of bool | None
```

```
type □ option = Some of □
     ↑          | None
    node
```

which can be instantiated into

int option, bool option, string option,

(int option) option, ...

Abstract Syntax Trees

Code

```
def fact(n):
    if n=0
        return 1
    else:
        return n * fact(n-1)
```

Multiple representations

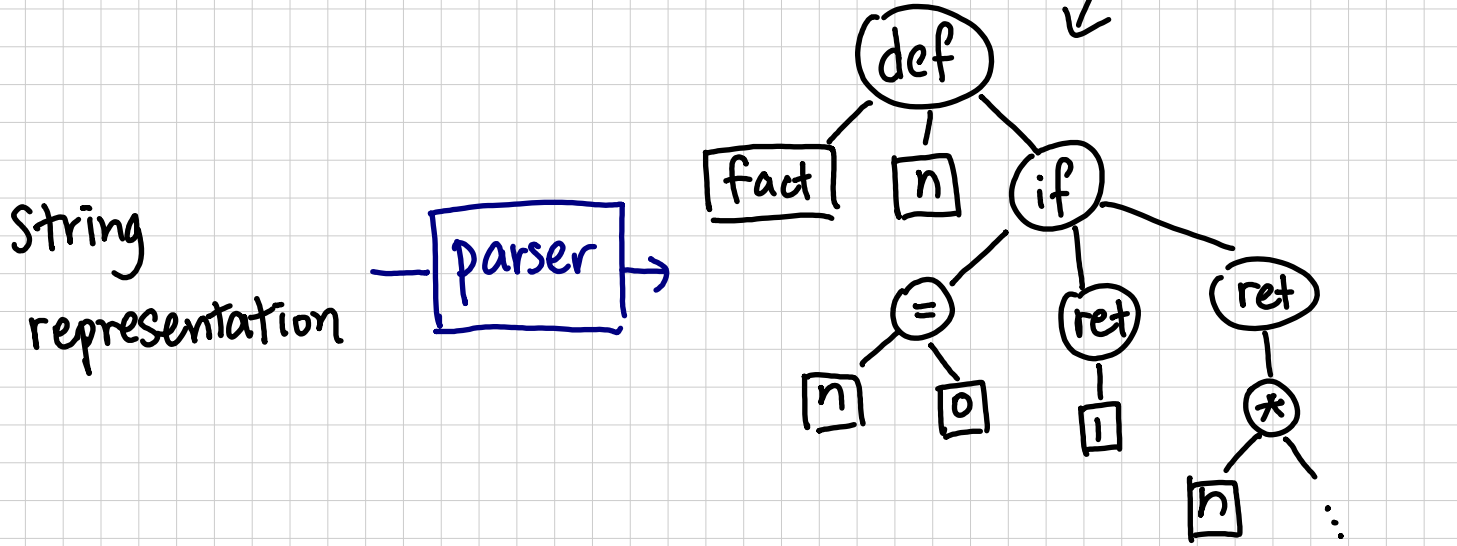
string representation

"def␣fact␣(n):\n␣if␣..."

Good for:
- + Coding using keyboard
- + storage

Bad for:
- - everything else

```
def fact(n):
    if n==0:
        return 1
    else:
        return n * fact(n-1)
```



abstract syntax tree (AST)

In CS162, we define language ASTs using context-free grammars (CFG).

A CFG can be easily represented using an OCaml data type.

Example

Language: simple calculator language

CFG:

```
expr = INT
     | expr + expr
     | expr - expr
     | expr * expr
```

OCaml type:

```
type expr = EInt of int
          | EAdd of expr * expr
          | ESub of expr * expr
          | EMul of expr * expr
```

The exact labels don't matter.

(isomorphic)

An equivalent encoding:

```
type op = Add | Sub | Mul
type expr = EInt of int
          | EArith of expr * op * expr
```

Recursion Recipe for CS162

(Free) 1. Identify the structure of the data:

e.g. 'a list = Nil | Cons of 'a * 'a list

(Free) 2. Decompose data into base case + recursive case

using pattern matching

(Easy) 3. Handle base case

4. Handle recursive case

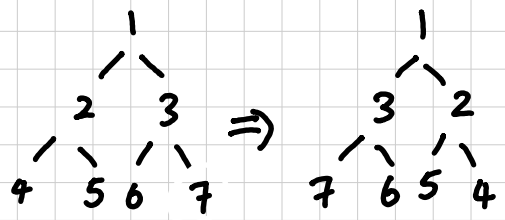
(Free) i) Call function on "smaller data"

ii) Assume that the result of i) is correct

compose solution to "bigger data"

only potential difficult part

Example: Mirror a binary tree.



(1) type tree = Leaf of int
| Node of int * tree * tree

(2) let rec mirror t =
match t with
| Nil → (3) Nil
| Cons(n, l, r) →

(4i)
let l' = mirror l in
let r' = mirror r in

(4ii)
Node(n, r', l')

Assume
mirror (3 / 7 6) = 3 / 6 7 = l'
mirror (2 / 5 4) = 2 / 4 5 = r'

Extended example:

Compute the cartesian product of two lists.

```
product [1; 2; 3]  ["a", "b", "c"]
= [ (1,"a") ; (1,"b") ; (1,"c") ;
    (2,"a") ; (2,"b") ; (2,"c") ;
    (3,"a") ; (3,"b") ; (3,"c") ]
```

Visually:

	a	b	c
1	1,a	1,b	1,c
2	2,a	2,b	2,c
3	3,a	3,b	3,c

① Identify the data structure:

```
type  list = Nil | Cons of  *
```

② Decompose the input:

```
let rec product (l1 : int list) (l2 : string list) : ((int * string) list) list =
    match l1 with
    | Nil -> (③ execercise for you)
    | Cons(h1, t1) ->
        ④i  let res = product t1 l2 in
            let first_row = row h1 l2 in
            first_row @ res
```

follow the recipe again!

Visually:

	a	b	c
1	1,a	1,b	1,c
2	2,a	2,b	2,c
3	3,a	3,b	3,c

= res

Let's define row 1 ["a"; "b"; "c"]

= [(1,"a"); (1,"b"); (1,"c")]

① 'a list = Nil | Cons of 'a * 'a list

② let rec row (x: int) (l: string list) : (int * string) list =

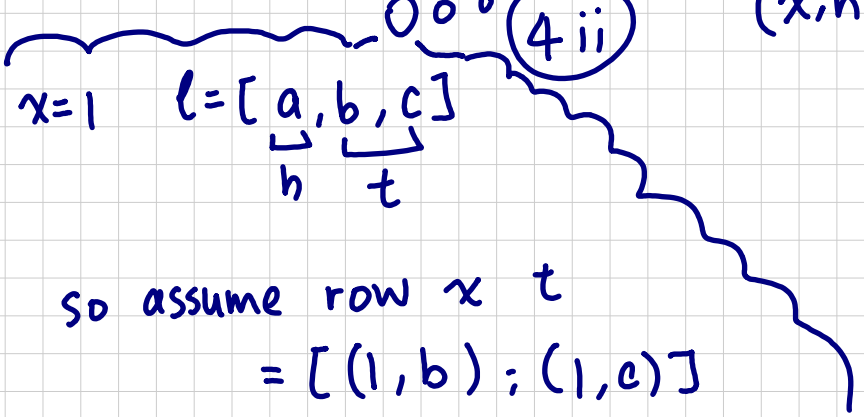
match l with

| Nil → ③ Nil

| Cons(h, t) →

④i let res = row x t in

④ii (x,h) :: t



so assume row x t

= [(1,b); (1,c)]

Want [(1,a); (1,b); (1,c)]

Extended example:

Compute the power set of a list.

[1; 2; 3] ⇒ [[]; [1]; [2]; [3]; [1;2]; [1;3]; [2;3]; [1;2;3]]

① list = ...

```
let rec power (l: int list) : (int list) list =
  ② match l with
    | Nil -> (③ exercise for you)
    | Cons(h, t) ->
        ④ let res = power t in
        ④ii let res' = insert h res in
           res @ res'
```

let's say l = [1; 2; 3] (h = 1, t = 2; 3)

So assume res = power [2; 3] = [[]; [2]; [3]; [2;3]]

Now you can define insert recursively using the recipe again (exercise)

Still missing this, but we can get this by inserting 1 at the beginning of every list in res.

want [[]; [2]; [3]; [2,3]; [1]; [1;2]; [1;3]; [1;2;3]] (res)