CS 320 : Functional Programming in Ocaml

(based on slides from David Walker, Princeton, Lukasz Ziarek, Buffalo and myself.)

Marco Gaboardi

MSC 116

gaboardi@bu.edu

Announcements

- New theory assignment posted yesterday, due Wednesday Oct 14, 11:59pm.

Assignment Project Exam Help

https://powcoder.com

In the previous class

Add WeChat powcoder

What is a Functional Language

A functional language:

- defines programs in a way similar to the one we useto define mathematical functions,
- avoids the use of mutable states (states that can change) in describing what a program should do.

In a functional language, the information is maintained by the computation.

Type Checking Rules

- Example rules:

(1) 0 : int (and similarly for any other integer constant n)

(2) "abc" : string (and similarly for any other string constant "…")

(3) if e1 : int and e2 : int
then e1 + e2 : int

(4) if e1 : int and e2 : int
then e1 * e2 : int

(5) if e1 : string and e2 : string
then e1 ^ e2 : string

(6) if e : int
then string_of_int e : string

- Using the rules:

2 : int and 3 : int.	(By rule 1)
Therefore, (2 + 3) : int	(By rule 3)
5 : int	(By rule 1)

Type Soundness

"Well typed programs do not go wrong"

Programming languages with this property have *sound* type systems. They are called *safe* languages.

Safe languages are generally *immune* to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc. (but not immune to all bugs!)

Safe languages: ML, Java, Python, …

Unsafe languages: C, C++, Pascal

Writing Functions Over Typed Data

Steps to writing functions over typed data:

1. Write down the function and argument names

2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   - *the argument types suggests how to do it*
5. Build new output values
   - *the result type suggests how you do it*
6. Clean up by identifying repeated patterns
   - define and reuse helper functions
   - your code should be elegant and easy to read

Types help structure your thinking about how to write programs.

Another Example


```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4      in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

-->

```
8
```

Moral: Let operates by *substituting* computed values for variables

Tuples

- To use a tuple, we extract its components
- General case:


```
let (id1, id2, …, idn) = e1 in e2
```


- An example:

```
let (x,y) = (2,4) in x + x + y
-->  2 + 2 + 4
-->  8
```

Unit

- Unit is the tuple with zero fields!

$\boxed{\text{() : unit}}$

  - the unit value is written with an pair of parens
  - there are no other values with this type!

- Why is the unit type and value useful?
- Every expression has a type:

```
(print_string "hello world\n")  :   unit
```

- Expressions executed for their *effect* return the unit value

Options

A value v has type t option if it is either:

- the value None, or
- a value Some v', and v' has type t

Options can signal there is no useful result to the computation

Example: we look up a value in a hash table using a key.

- If the key is present, return Some v where v is the associated value
- If the key is not present, we return None

Tail recursion

- Tail recursion is the idea to write code of recursive functions in a way that the recursive call happen in the tail, i.e. as the last operation.

- Often it is implemented using some helper function that accumulates the result while performing the recursion.

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

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

3 + 4 : int

add (3 + 4) : int -> int

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

Curried Functions

Currying: *verb. gerund or present participle*

(1) to prepare or flavor with hot-tasting spices

(2) to encode a multi-argument function using nested, higher-order functions.

(1)

(2)

```
fun x -> (fun y -> x+y) (* curried *)
fun x y -> x + y        (* curried *)
fun (x,y) -> x+y        (* uncurried *)
```

Factoring Code in OCaml

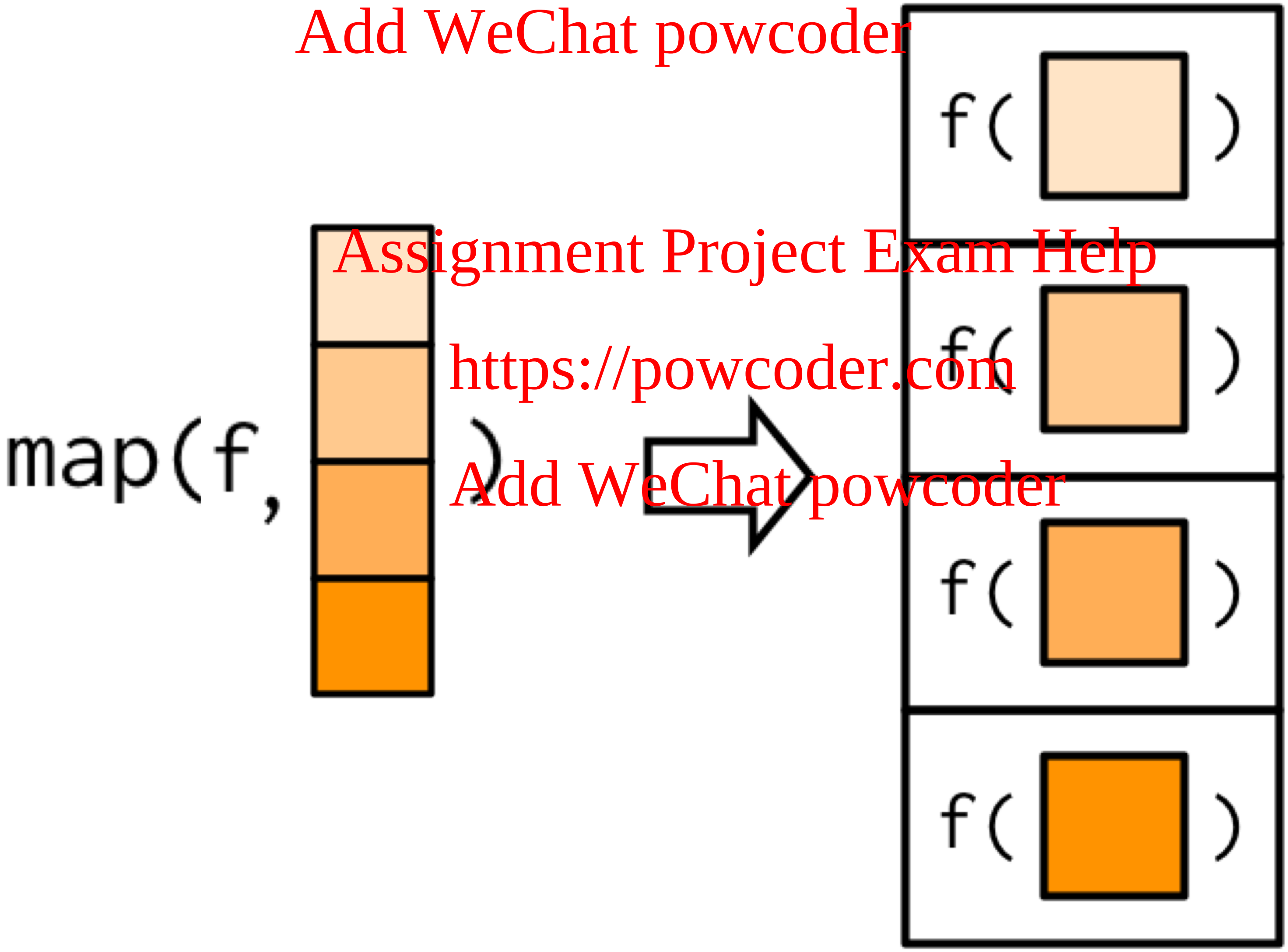
Consider these definitions:


```
let rec inc_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd+1)::(inc_all tl)
```



```
let rec square_all (xs:int list) : int list =
  match xs with
  | [] -> []
  | hd::tl -> (hd*hd)::(square_all tl)
```

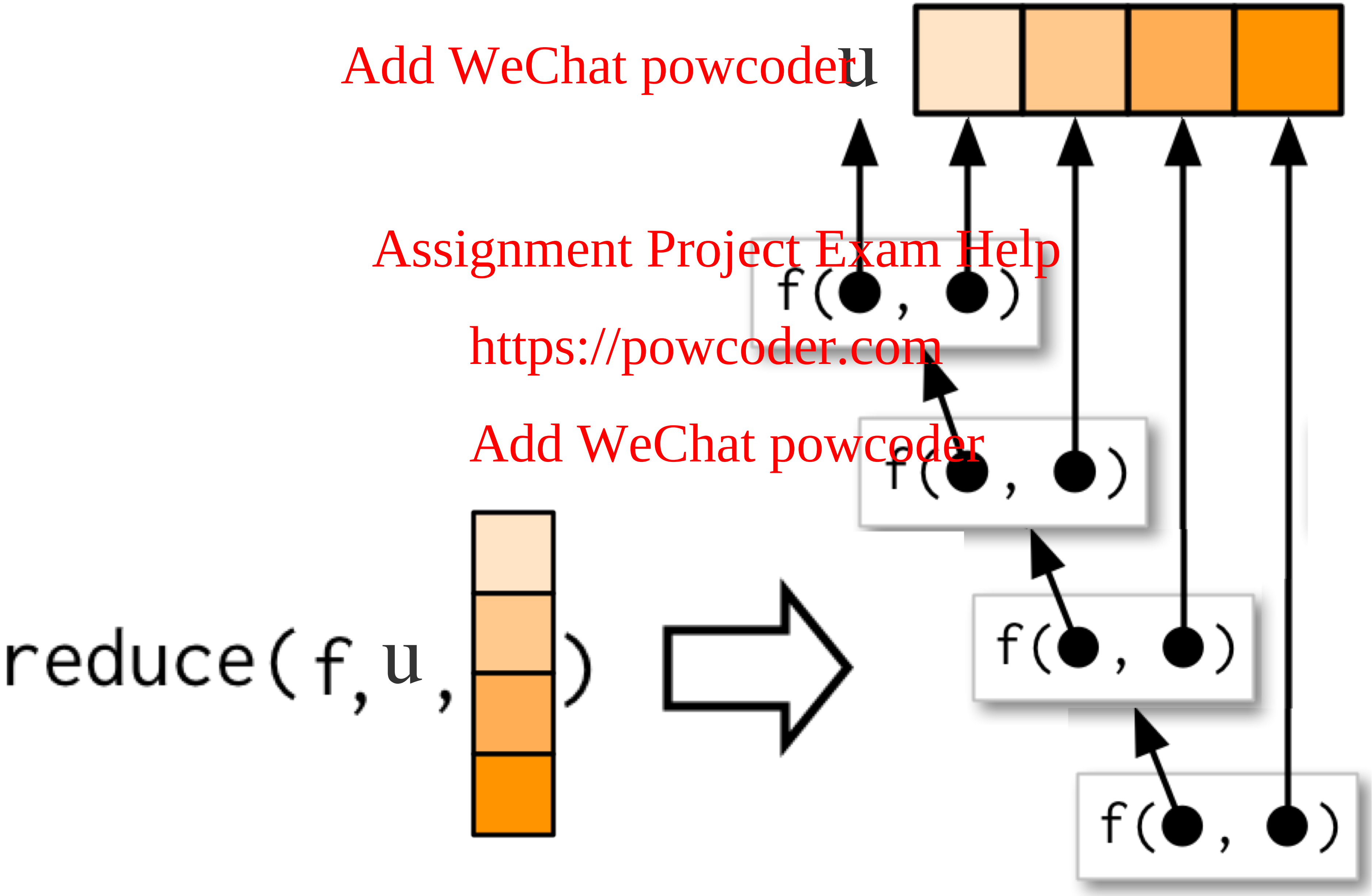

The code is almost identical – factor it out!

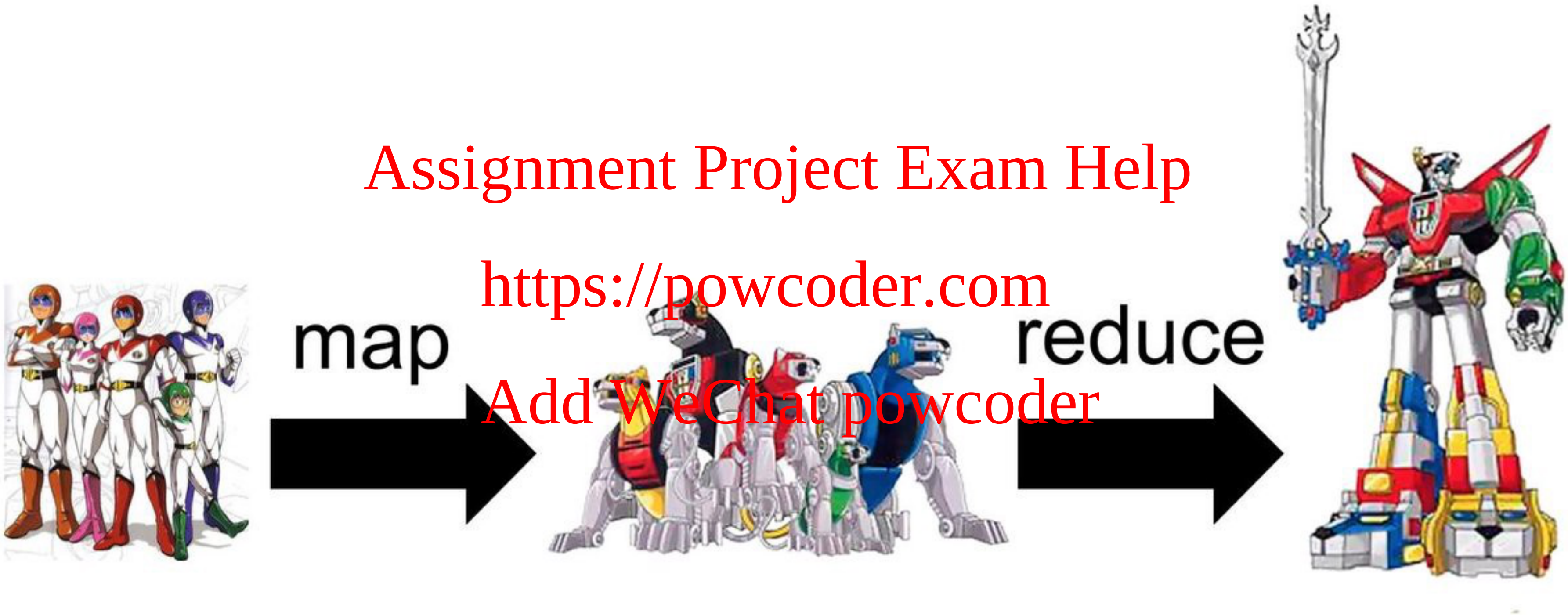


Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder


u
f(,)
f(,)
f(,)
f(,)
reduce(f, u,)



Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Type of the undecorated map?


```
let rec map f xs =
  match xs with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)

map : ('a -> 'b) -> 'a list -> 'b list
```

How about reduce?


```
let rec reduce (f:'a -> 'b -> 'b) (u:'b) (xs: 'a list) : 'b =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)
```


What's the most general type of reduce?


```
('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

And this one?


```
let rec reduce f u xs =
  match xs with
  | [] -> u
  | hd::tl -> f hd (reduce f u tl)

let mystery2 g =
   reduce (fun a b -> (g a)::b) []

let rec mystery2 g xs =
  match xs with
  | [] -> []
  | hd::tl -> (g hd)::(mystery2 g tl) map!
```

Learning goals for today

- More exercises on Inductive Data Types

Inductive Data Types

Inductive data types

- We can use data types to define inductive data
- A binary tree is:
  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree

Inductive data types

- We can use data types to define inductive data
- A binary tree is:

  - a Leaf containing no data
  - a Node containing a key, a value, a left subtree and a right subtree


```
type key = string
type value = int

type tree =
   Leaf
 | Node of key * value * tree * tree
```

Inductive data types


```
type key = int
type value = string

type tree =
  Leaf
| Node of key * value * tree * tree
```



```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
   | Leaf -> Node (k, v, Leaf, Leaf)
   | Node (k', v', left, right) ->
       if k < k' then
         Node (k', v', insert left k v, right)
       else if k > k' then
         Node (k', v', left, insert right k v)
       else
         Node (k, v, left, right)
```

```
type ('key, 'val) tree =
    Leaf
  | Node of 'key * 'val * ('key, 'val) tree * ('key, 'val) tree
```



```
type 'a stree = (string, 'a) tree
```



```
type sitree = int stree
```


General form:

definition:

```
type 'x f = body
```

use:

```
arg f
```

A more conventional notation would have been (but is not ML):

definition:

```
type f x = body
```

use:

```
f arg
```

Write a function taking in input a tree and returning the number of leaves.


```
type tree = Leaf | Node of (int * string * tree * tree)
```



```
let rec leaves (t:tree) : int =
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Write a function taking in input a tree and returning the number of leaves.


```
type tree = Leaf | Node of (int * string * tree * tree)


let rec leaves (t:tree) : int =
  match t with
   |Leaf -> 1
   | Node (_,_,t1,t2) -> (leaves t1) + (leaves t2)
```

Write a data type for binary trees with internal nodes and labeled by Booleans.

Write a fold function for this data type.


```
type `a btree =
```



```
let rec fold
```

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

Write a data type for binary trees with internal nodes labeled by elements of an arbitrary type.

Write a fold function for this data type.

```
type `a btree = Leaf | Node of (`a * (`a btree) * (`a btree))
```



```
let rec fold  (f: `a -> `b -> `b -> `b) (a:`b)
              (t:`a btree) : `b=
match t with
|Leaf -> a
|Node (n,l,r) -> f n (fold f a l) (fold f a r)
```

Write a data type for a stack of integers.

Write a function push and a function pop


```
type stack =
```



```
let push

let pop
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder

Write a data type for a stack of integers.

Write a function push and a function pop

```
type stack = Empty | Cons of (int * stack)
```



```
let push (i:int) (s:stack) : stack = Cons (i,s)
```



```
let pop (s:stack) :stack option = match s with
Empty -> None
| Cons (_,xs) -> Some xs;;
```

Input/Output on files in OCaml

Input and Output Channels

The normal way of opening a file in OCaml returns a **channel**. There are two kinds of channels:

- channels that write to a file: type `out_channel`
- channels that read from a file: type `in_channel`

Four operations that will be useful are:

- Open input file: `open_in: string -> in_channel`
- Open out file: `open_out: string -> out_channel`
- Close input file: `close_in: in_channel -> unit`
- Close input file: `close_out: out_channel -> unit`

If you want to use a channel, you can use `let`, as usual.

Discarding an expression

Often we may need to discard an expression

- This happens often with `unit`, when it is returned and we don't need it.

An easy way to discard an expression is by using let with a variable that does not appear in the body:


```
let x = printf "%s/n" str in 3+2
```


In this case, we can also use underscore to avoid giving a name to this variable:


```
let _ = printf "%s/n" str in 3+2
```

This is often abbreviated in ocaml using a semicolon:

```
printf "%s/n" str; 3+2
```

Reading a line


```
let read_line (inc:in_channel): string option =
match input_line inc with
| l -> Some l
| exception End_of_file -> None
```


Writing a line

```
Printf.fprintf ouc "%s\n" str
```

The types need to match

An example – copying one line:


```
let ic=open_in ``tmp.in'' in
let oc=open_out ``tmp.ou'' in
let l=read_line ic in
let _ = match l with
       Some s -> Printf.fprintf oc ``%s/n'' s in
      |None -> ()
let _= close_in ic in
let _= close_out oc in()
```

Practice Problems

Using map, write a function that takes a list of pairs of integers, and produces a list of the sums of the pairs.

- e.g., list_add [(1,3); (4,2); (3,0)] = [4; 6; 3]
- Write list_add directly using reduce.

Using map, write a function that takes a list of pairs of integers, and produces their quotient if it exists.

- e.g., list_div [(1,3); (4,2); (3,0)] = [Some 0; Some 2; None]
- Write list_div directly using reduce.

Using reduce, write a function that takes a list of optional integers, and filters out all of the None's.

- e.g., filter_none [Some 0; Some 2; None; Some 1] = [0;2;1]
- Why can't we directly use filter? How would you generalize filter so that you can compute filter_none? Alternatively, rig up a solution using filter + map.

Using reduce, write a function to compute the sum of squares of a list of numbers.

- e.g., sum_squares = [3,5,2] = 38

Summary

- Exercise on higher-order functions
- Data Types
- Inductive Data Types