

CSE 130 : Spring 2011

Programming Languages

Lecture 7:
Higher-Order Functions

Ranjit Jhala
UC San Diego



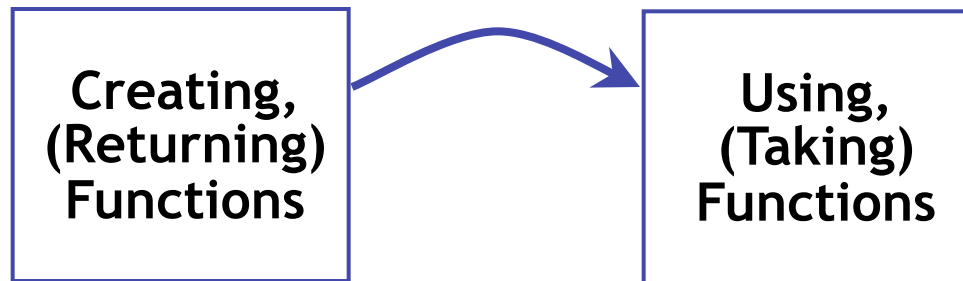
News

- PA4 is up
 - Due 5/6
 - Week **AFTER** midterm
- Midterm 4/28 (?)
 - In class, open book etc.
 - Practice materials on Webpage
 - Depends on PA4 ...

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*

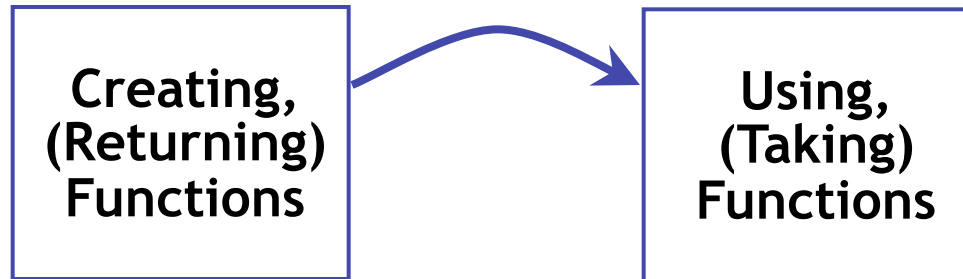


Useful if parameterized functions can be passed to, hence used/called by other functions...

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*



Why take functions as input ?

Why take functions as input ?

```
let rec evens 1 =
```

Why take functions as input ?

```
let rec evens l =  
  match l with  
    [] -> []  
  | h::t -> if is_even h then h::(evens t) else evens t
```

Why take functions as input ?

```
let rec evens l =  
  match l with  
    [] -> []  
  | h::t -> if is_even h then h::(evens t) else evens t
```

```
let rec lessers x l =
```

Why take functions as input ?

```
let rec evens l =  
  match l with  
    [] -> []  
  | h::t -> if is_even h then h::(evens t) else evens t
```

```
let rec lessers x l =  
  match l with  
    [] -> []  
  | h::t -> if h < x then h::(lessers x t) else lessers x t
```

Why take functions as input ?

```
let rec evens l =  
  match l with  
    [] -> []  
  | h::t -> if is_even h then h::(evens t) else evens t
```

```
let rec lessers x l =  
  match l with  
    [] -> []  
  | h::t -> if h<x then h::(lessers x t) else lessers x t
```

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

Factoring and Reuse

```
let rec lessers x l =  
  match l with  
    [] -> []  
  | h::t -> if h<x then h::(lessers x t) else lessers x t
```

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

```
let lessers x l =  
  filter (fun i -> i<x) l
```

Factoring and Reuse

```
let rec lessers x l =  
  match l with  
    [] -> []  
  | h::t -> if h<x then h::(lessers x t) else lessers x t
```

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

“Factor” code:

- Generic pattern
- Specific instance

```
let lessers x l =  
  filter (fun i -> i<x) l
```

Factoring and Reuse

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

“Factor” code:

- Generic pattern
- Specific instance

```
let evens l =  
  filter is_even l
```

Factoring and Reuse

```
let rec evens l =  
  match l with  
    [] -> []  
  | h::t -> if is_even h then h::(evens t) else evens t
```

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

“Factor” code:

- Generic pattern
- Specific instance

```
let evens l =  
  filter is_even l
```

Encoding Patterns as functions

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t)  
             else (filter f t);;
```

Encoding Patterns as functions

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t)  
             else (filter f t);;
```

```
let neg f = fun x -> not (f x)  
let partition f l = (filter f l, filter(neg f) l)
```

Encoding Patterns as functions

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t)  
             else (filter f t);;
```

```
let neg f = fun x -> not (f x)  
let partition f l = (filter f l, filter(neg f) l))
```

filter, neg, partition: higher-order functions

- Take a **any** tester as argument!

Iteration Pattern

```
let rec listUppercase xs =  
  match xs with  
    []      -> []  
  | h::t -> (uppercase h) :: (listUppercase t)
```

```
let rec listSquare xs =  
  match xs with  
    []      -> []  
  | h::t -> (h * h) :: (listSquare t)
```

```
let addPair (x,y) = x + y  
let rec listAddPair xs =  
  match l with  
    []      -> []  
  | (hx,hy) :: t -> (addPair (hx,hy)) :: (listAddPair t)
```

Iteration Pattern

```
let rec listUppercase xs =  
  match xs with  
    []      -> []  
  | h::t -> (uppercase h)::(listUppercase t)
```

```
let rec listSquare xs =  
  match xs with  
    []      -> []  
  | h::t -> (h * h)::(listSquare t)
```

```
let addPair (x,y) = x + y  
let rec listAddPair xs =  
  match l with  
    []      -> []  
  | (hx,hy)::t -> (addPair (hx,hy))::(listAddPair t)
```

Iteration Pattern

```
let rec listUppercase xs =  
  match xs with  
    []      -> []  
  | h::t    -> (uppercase h)::(listUppercase t)
```

```
let rec listSquare xs =  
  match xs with  
    []      -> []  
  | h::t    -> (h * h)::(listSquare t)
```

```
let addPair (x,y) = x + y  
let rec listAddPair xs =  
  match xs with  
    []      -> []  
  | (hx,hy)::t -> (addPair (hx,hy))::(listAddPair t)
```

Iteration Pattern

```
let rec listUppercase xs =  
  match xs with  
    []      -> []  
  | h::t    -> (uppercase h) :: (listUppercase t)
```

Iteration Pattern

```
let rec map f l =  
  match l with  
    [] -> []  
  | (h::t) -> (f h)::(map f t)
```

uppercase

Iteration Pattern

```
let rec map f l =  
  match l with  
  [] -> []  
  | (h::t) -> (f h) :: (map f t)
```

uppercase

Iteration Pattern

```
let rec map f l =  
  match l with  
  [] -> []  
  | (h::t) -> (f h)::(map f t)
```

uppercase

```
let listUpperCase l = map uppercase
```

1

Iteration Pattern

```
let rec map f l =  
  match l with  
  [] -> []  
  | (h::t) -> (f h)::(map f t)
```

uppercase

```
let listUpperCase l = map uppercase
```

```
let listSquare l = map (fun x -> x*x) l
```

Iteration Pattern

```
let rec map f l =  
  match l with  
    [] -> []  
  | (h::t) -> (f h) :: (map f t)
```

uppercase

```
let listUpperCase l = map uppercase
```

```
let listSquare l = map (fun x -> x*x) l
```

```
let listAddpair l = map (fun (x,y) -> x+y) l
```

Higher-order functions: map

```
let rec map f l =  
  match l with  
  [] -> []  
| (h::t) -> (f h) :: (map f t)
```

$('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

Type says it all !

- Applies “f” to each element in input list
- Makes a list of the results

Factoring Iteration w/ “map”

```
let rec map f l =  
  match l with  
    [] -> []  
  | (h::t) -> (f h) :: (map f t)
```

Factoring Iteration w/ “map”

```
let rec map f l =  
  match l with  
    [] -> []  
  | (h::t) -> (f h) :: (map f t)
```

“Factored” code:

- Reuse iteration template
- Avoid bugs due to repetition
- Fix bug in one place !

Another pattern: Accumulation

```
let max x y = if x > y then x else y ;
let listMax l =
  let rec help cur l =
    match l with
    | [] -> cur
    | h::t -> help (max cur h) t
  in
    helper 0 l;;
```

Another pattern: Accumulation

```
let max x y = if x > y then x else y ;
let listMax l =
  let rec help cur l =
    match l with
    [] -> cur
    | h::t -> help (max cur h) t
  in
    helper 0 l;;
```

```
let concat l =
  let rec help cur l =
    match l with
    [] -> cur
    | h::t -> help (cur^h) t
  in
    helper "" l;;
```

Whats the pattern ?

Whats the pattern ?

Whats the pattern ? **Tail Rec ?**

```
Let rec fold f cur l =  
  case l of  
    [] -> cur  
  | h::t -> fold f (f cur h) t
```

What is: `fold f base [v1;v2;...;vn]` ?

```
f (... ( f (
                                     , v3) , vn)
      f (base, v1)
    f ( f (
        , v2)
      , v3)
  f (... (
          , vn)
```

Whats the pattern ?

;

.

.

Whats the pattern ?

Tail Rec ?

Examples of fold

```
let listMax =
```

Currying! This is a function!

```
fold max 0
```

```
let concat =
```

Currying! This is a function!

```
fold (^) ""
```

Pick correct
base case!

```
let multiplier =
```

Examples of fold

```
let listMax =
```

```
let concat =
```

```
let multiplier =
```

Examples of fold

What does this do ?

```
let f l = fold (::) [] l
```

Examples of fold

```
let f l = fold (::) [] l
```

Funcs taking/returning funcs

Identify common computation “patterns”

- Filter values in a set, **list**, tree ...
- Iterate a function over a set, **list**, tree ...
map
- Accumulate some value over a collection

fold

Pull out (factor) “common” code:

- Computation Patterns
- Re-use in many different situations

Another fun function: “pipe”

Another fun function: “pipe”

```
let pipe x f = f x
```

Another fun function: “pipe”

```
let pipe x f = f x
```

```
let (|>) x f = f x
```

Another fun function: “pipe”

```
let pipe x f = f x
```

```
let (|>) x f = f x
```

Compute the sum of squares of numbers in a list ?

Another fun function: “pipe”

```
let pipe x f = f x
```

```
let (|>) x f = f x
```

Compute the sum of squares of numbers in a list ?

```
let sumOfSquares xs =  
  xs |> map (fun x -> x * x)  
  |> fold_left (+) 0
```

Another fun function: “pipe”

```
let pipe x f = f x
```

```
let (|>) x f = f x
```

Compute the sum of squares of numbers in a list ?

```
let sumOfSquares xs =  
  xs |> map (fun x -> x * x)  
  |> fold_left (+) 0
```

Tail Rec ?

Funcs taking/returning funcs

Identify common computation “patterns”

- Filter values in a set, list, tree ...
- Convert a function over a set, list, tree ... 
- Iterate a function over a set, list, tree ...
- Accumulate some value over a collection 

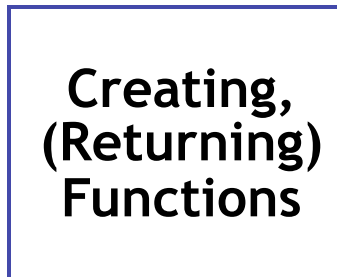
Pull out (factor) “common” code:

- Computation Patterns
- Re-use in many different situations

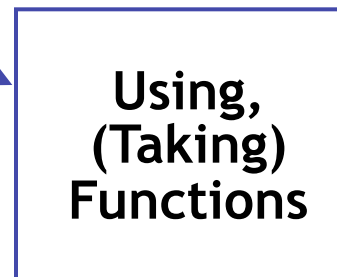
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*



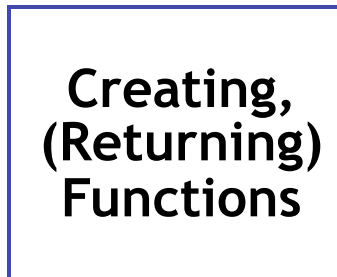
*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*



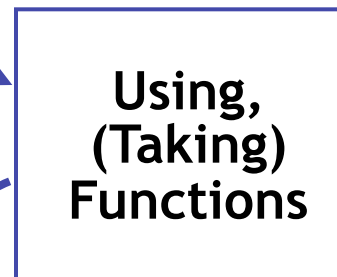
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?

*Parameterized,
similar functions
(e.g. Testers)*



*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*

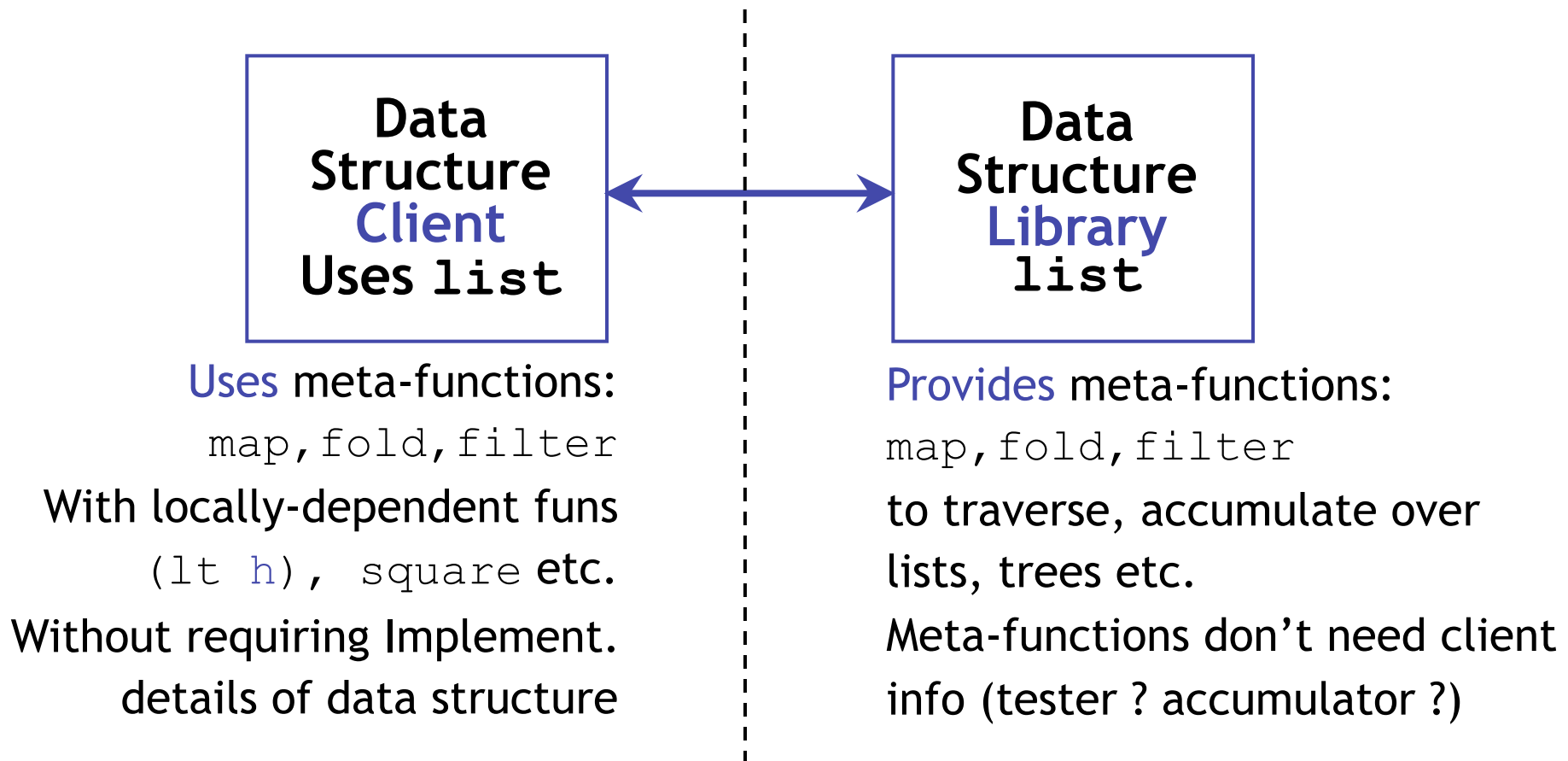


*Compose Functions:
Flexible way to build
Complex functions
from primitives.*

Funcs taking/returning funcs

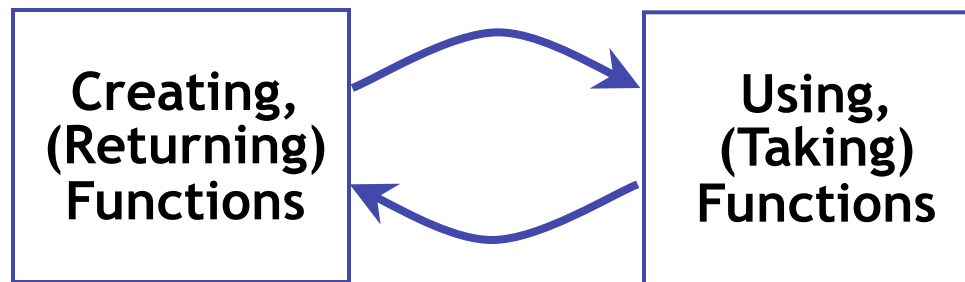
Higher-order funcs enable **modular** code

- Each part only needs **local** information



Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits ?



Everywhere:

- Javascript,
- Google/Mapreduce, Yahoo/Hadoop,
- C++0x

Parametric types

aka: what's up with those 'a' ?

What is the deal with 'a ?

These meta-functions have strange types:

map: $('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

filter: $('a \rightarrow \text{bool}) \rightarrow 'a \text{ list} \rightarrow 'a \text{ list}$

Why ?

Polymorphism

- Poly = *many*, morph = *kind*

```
let swap (x, y) = (y, x)
```

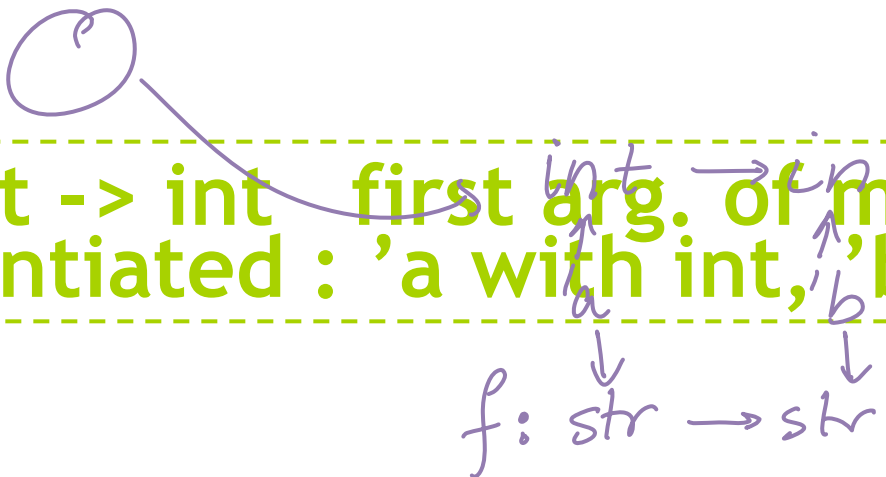
 $'a * 'b \rightarrow 'b * 'a$

- *'a* and *'b* are type variables!
- For-all types: *For all* $'a, 'b: 'a * 'b \rightarrow 'b * 'a$
- *'a, 'b* can be instantiated with any type:
 - w/ *int, string* : $int * string \rightarrow string * int$
 - w/ *char, int list* : $char * int\ list \rightarrow int\ list * char$
 - w/ *int $\rightarrow$ int* , *bool* : $(int \rightarrow int) * bool \rightarrow bool * (int \rightarrow int)$

Instantiation at Use

map:

$(\text{'a} ! \text{'b}) ! \text{'a list} ! \text{'b list}$

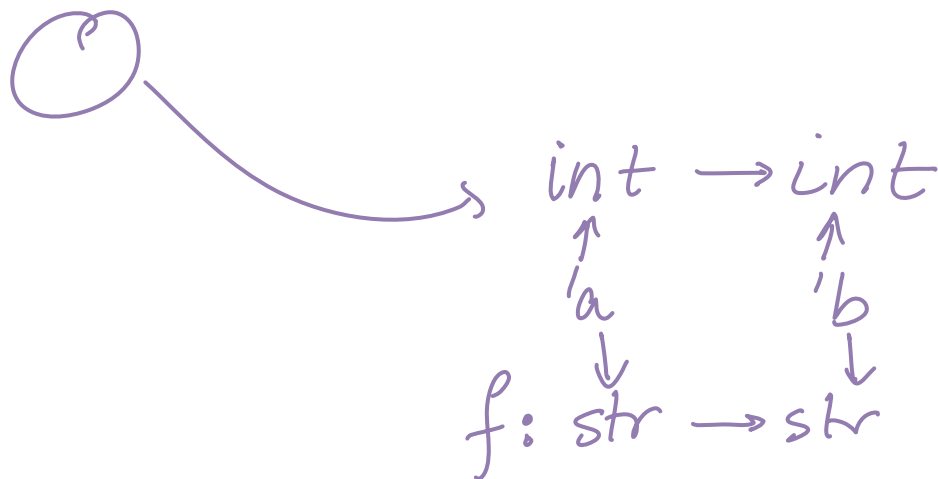

f : int -> int first arg. of map: a->b
Instantiated : 'a with int, 'b with int
f : str -> str


f : string -> string first arg. of map: a->b
Instantiated : 'a with str, 'b with str

Instantiation at Use

map:

$(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$



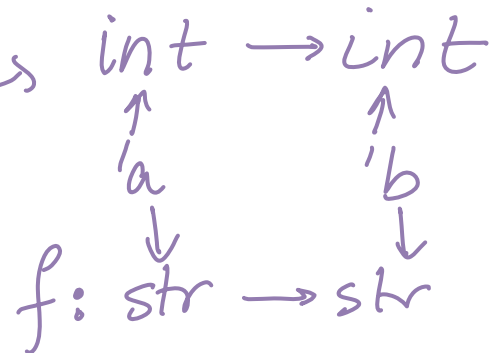
`map string->int`

Instantiation at Use

map: $(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$

```
let f x = x + 10;;
```

```
let fm = map f;;
```



`map string -> int`

Instantiation at Use

map: $(\text{'a} \rightarrow \text{'b}) \rightarrow \text{'a list} \rightarrow \text{'b list}$

```
let f x = x + 10;;  
let fm = map f;;
```

$\text{int} \rightarrow \text{int}$
 $\uparrow \quad \uparrow$
 $\text{'a} \quad \text{'b}$
 $\downarrow \quad \downarrow$
 $f: \text{str} \rightarrow \text{str}$

```
let f x = x ^ " like" ;;  
let fm = map f ["cat"; "dog"; "burrito"];;
```

$\text{map string} \rightarrow \text{int}$

Instantiation at Use: be careful

map: $('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

```
let f x = x ^ " like";;
```

```
let fm = (map f) [1;2;3;4];;
```

int list

string list $\rightarrow$ string list

f : string -> string first arg. of map: a->b

Instantiated : 'a with str, 'b with str'

So, list must be 'a list = string list !

Instantiation at Use: be careful

map: $(a \rightarrow b) \rightarrow a \text{ list} \rightarrow b \text{ list}$

```
let f x = x ^ " like";;
```

```
let fm = (map f) [1;2;3;4];;
```

String list $\rightarrow$ string list

Polymorphic ML types

- Poly = *many*, morph = *kind*

Polymorphic ML types

- Poly = *many*, morph = *kind*
- Possible ML types:

Polymorphic ML types

- Poly = *many*, morph = *kind*
- Possible ML types:

$tv = 'a \mid 'b \mid 'c \mid \dots$

$T = int \mid bool \mid string \mid char \mid \dots$

$T_1 * T_2 * \dots T_n \mid T_1 \rightarrow T_2 \mid tv$

Polymorphic ML types

- Poly = *many*, morph = *kind*
- Possible ML types:

$tv = 'a \mid 'b \mid 'c \mid \dots$

$T = int \mid bool \mid string \mid char \mid \dots$

$T_1 * T_2 * \dots T_n \mid T_1 \rightarrow T_2 \mid tv$

- Implicit for-all at the “*left*” of all types
 - Never printed out

Polymorphic ML types

- Poly = *many*, morph = *kind*
- Possible ML types:

$tv = 'a \mid 'b \mid 'c \mid \dots$

$T = int \mid bool \mid string \mid char \mid \dots$

$T_1 * T_2 * \dots T_n \mid T_1 \rightarrow T_2 \mid tv$

- Implicit for-all at the “*left*” of all types
 - Never printed out

map:

$('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

Polymorphic ML types

- Poly = *many*, morph = *kind*
- Possible ML types:

$tv = 'a \mid 'b \mid 'c \mid \dots$

$T = int \mid bool \mid string \mid char \mid \dots$

$T_1 * T_2 * \dots T_n \mid T_1 \rightarrow T_2 \mid tv$

- Implicit for-all at the “*left*” of all types
 - Never printed out

map: *For all 'a , 'b : ('a -> 'b) -> 'a list -> 'b list*

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 !

Not just functions ...

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

Not just functions ...

- Data types are also polymorphic!

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

Not just functions ...

- Data types are also polymorphic!

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

Not just functions ...

- Data types are also polymorphic!

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

Not just functions ...

- Data types are also polymorphic!

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

- Type is instantiated for each use:

Not just functions ...

- 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

Not just functions ...

- 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

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

Datatypes with many type variables

- Multiple type variables

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

Datatypes with many type variables

- Multiple type variables

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

Datatypes with many type variables

- Multiple type variables

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

Datatypes with many type variables

- Multiple type variables

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

- Type is instantiated for each use:

Leaf("joe",1) :

Leaf("william",2) :

Node(...,...) :

Node(Leaf("joe",1),Leaf(3.14, "pi")):

Datatypes with many type variables

- Multiple type variables

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

- Type is instantiated for each use:

Leaf("joe",1) : (string,int) tree

Leaf("william",2) : (string,int) tree

Node(...,...) : (string,int) tree

Node(Leaf("joe",1),Leaf(3.14, "pi"))

Polymorphic Data Structures

	<i>'a list</i>
<i>key, data</i>	<i>('a , 'b) tree</i>
	<i>('a , 'b) hashtable ...</i>

Polymorphic Data Structures

'a list

('a , 'b) tree

('a , 'b) hashtable ...

Polymorphic Data Structures

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

'a list

('a , 'b) tree

('a , 'b) hashtable ...

Polymorphic Data Structures

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

'a list

('a , 'b) tree

('a , 'b) hashtable ...

Polymorphic Data Structures

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

'a list

('a , 'b) tree

('a , 'b) hashtable ...

Polymorphic Data Structures

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

'a list

('a , 'b) tree

('a , 'b) hashtable ...

- Appropriate type instantiated at use
 - No *unsafe casting* as in C/C++/Java

Polymorphic Data Structures

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

'a list

('a , 'b) tree

('a , 'b) hashtable ...

- Appropriate type instantiated at use
 - No *unsafe casting* as in C/C++/Java
- *Static type checking catches errors* early
 - Cannot add *int* key to *string* hashtable

Polymorphic Data Structures

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

'a list

('a , 'b) tree

('a , 'b) hashtable ...

- Appropriate type instantiated at use
 - No *unsafe casting* as in C/C++/Java
- *Static type checking catches errors* early
 - Cannot add *int* key to *string* hashtable

Polymorphic Data Structures

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

'a list

('a , 'b) tree

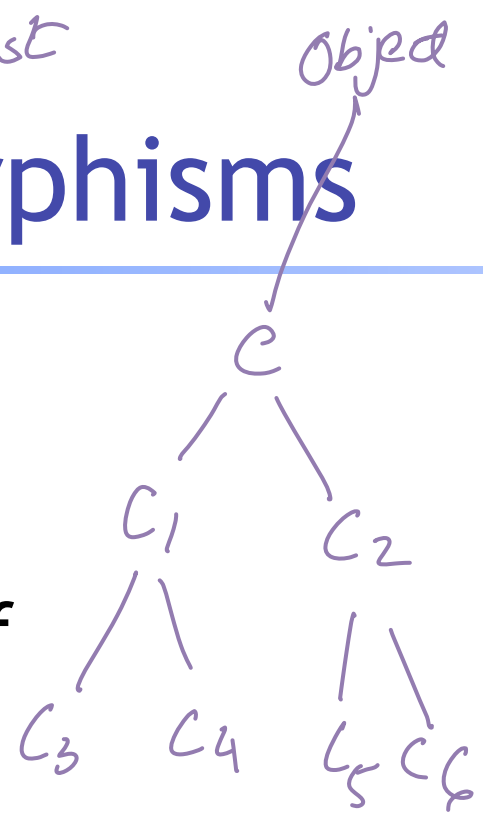
('a , 'b) hashtable ...

- Appropriate type instantiated at use
 - No **unsafe casting** as in C/C++/Java
- **Static type checking catches errors** early
 - Cannot add **int** key to **string** hashtable
- **Generics**: in Java, C#, VB (borrowed from ML)

List_rev : Object list $\rightarrow$ Object list
'a list $\rightarrow$ 'a list

Other kinds of polymorphisms

- That was OCaml...
- But what about other kinds of polymorphisms..



foo(int x, int y) { ... }

foo(float x, float y) { ... }

"overloading"

Other kinds of polymorphisms

$\text{list}\langle T \rangle \text{ reverse } (\text{list}\langle T \rangle \text{ xs}) \{ \dots \}$

Other kinds of polymorphisms

- Sub-type polymorphism

$\text{List}(T) \text{ reverse}(\text{list}(T) \text{ xs}) \{ \dots \}$

Other kinds of polymorphisms

- Sub-type polymorphism

`void f(Shape s)`

`List<T> reverse(list<T> xs) { ... }`

Other kinds of polymorphisms

- Sub-type polymorphism

`void f(Shape s)`

- Can pass in any sub-type of Shape

`List<T> reverse(list<T> xs) { ... }`

Other kinds of polymorphisms

- Sub-type polymorphism

`void f(Shape s)`

- Can pass in any sub-type of Shape

`List<T> reverse(list<T> xs) { ... }`

Other kinds of polymorphisms

- Sub-type polymorphism

`void f(Shape s)`

- Can pass in any sub-type of Shape

- Parametric polymorphism

`List<T> reverse(list<T> xs) { ... }`

Other kinds of polymorphisms

- Sub-type polymorphism

```
void f(Shape s)
```

- Can pass in any sub-type of Shape

- Parametric polymorphism

```
void proc_elems(list[T])
```

$\text{list}(T) \text{ reverse}(\text{list}(T) \text{ xs}) \{ \dots \}$

Other kinds of polymorphisms

- Sub-type polymorphism

```
void f(Shape s)
```

- Can pass in any sub-type of Shape

- Parametric polymorphism

```
void proc_elems(list[T])
```

- can pass in ANY T

list<T> reverse(list<T> xs) { ... }

Other kinds of polymorphisms

- Sub-type polymorphism

`void f(Shape s)`

- Can pass in any sub-type of Shape

- Parametric polymorphism

`void proc_elems(list[T])`

- can pass in ANY T
- this is the kind in OCaml!

list<T> reverse(list<T> xs) { ... }

Other kinds of polymorphisms

Other kinds of polymorphisms

- Bounded polymorphism

Other kinds of polymorphisms

- Bounded polymorphism

- Like parametric, except can provide a bound

```
void proc_elems(list[T]) T extends Printable
```

- Hey... isn't this subtype polymorphism?

- No, for example:

```
bool ShapeEq(T a, T b) T extends Shape
```

- Can call on

- (Rect, Rect)

- (Circle, Circle)

- But not (Rect, Circle)

Summary of polymorphism

- Subtype
- Parametric
- Bounded = Parametric + Subtype
 - In Java/C#

Back to OCaml

- Polymorphic types allow us to reuse code
- However, not always obvious from staring at code
- But... Types never entered w/ program!