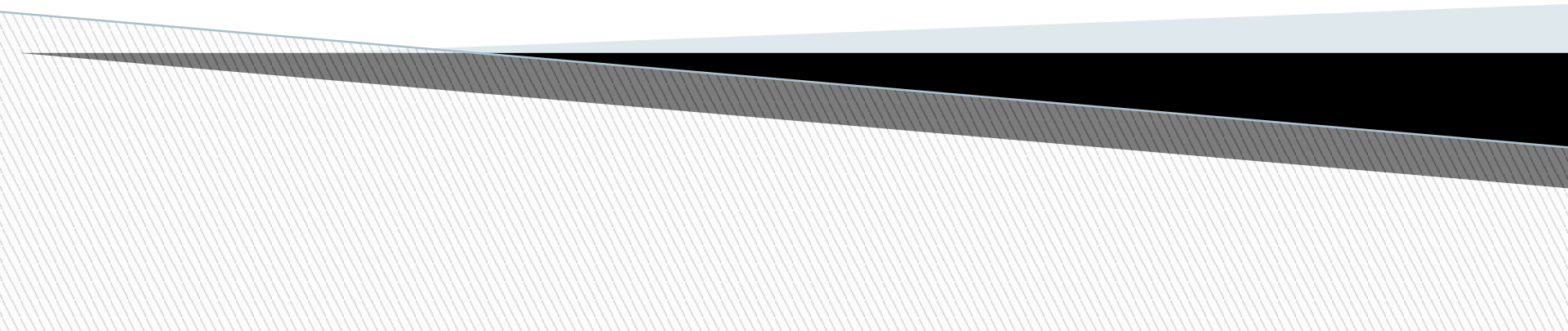
Higher-Order Functions



Higher-Order Functions

- Higher-order function/ Functional forms
  - it takes a function as an argument or
  - returns a function as a result

- **Example**
- **-call/cc**
  - Parameter will be a closure with value of PC and referencing environment
  - Transfers control to referencing environment
- **for-each**
  - Takes as argument a function and a list
- **apply**
  - Takes as argument a function and a list

map

- Takes a function and a sequence of list as arguments
- There must be as many list as arguments
- Lists must be of same length
- Map calls the function on corresponding sets of elements from the list

- (map * ’(2 4 6) ’(3 5 7)) $\Longrightarrow$ (6 20 42)

- for-each
  - Executed for side effects
  - Has implementation dependent return value
- Map
  - Purely functional
  - Returns a list having values returned by the function

fold

- Method of reducing a sequence of terms down to a single term
- To fold elements of a list together

```
(define fold (lambda (f i l)
 (if (null? l) i ;
 (f (car l) (fold f i (cdr l))))))

    (fold + 0 ’(1 2 3 4 5))
    (fold * 1 ’(1 2 3 4 5))
```

- A common use of higher order function is to build new functions from existing ones

```
(define total (lambda (l) (fold + 0 l)))

(total ’(1 2 3 4 5)) ⟹ 15

(define total-all (lambda (l)
(map total l)))

(total-all ’((1 2 3 4 5)
(2 4 6 8 10)
(3 6 9 12 15))) ⟹ (15 30 45)
```

Currying

- Replace multiargument function with a function
  - that takes a single arg &
  - returns a function that expects remaning args

(define curried-plus (lambda (a) (lambda (b) (+ a b))))
((curried-plus 3) 4) ⟹ 7

- **Alternative way**

(define plus-3 (curried-plus 3))
(plus-3 4) ⟹ 7

- Ability to pass partially applies function to a higher order function

(define curried-plus (lambda (a) (lambda (b) (+ a b))))

(map (curried-plus 3) ’(1 2 3)) =⇒ (4 5 6)

ML and its Descendants

- Functions in ML
- fun plus (a, b) : int = a + b;
- ==> val plus = fn : int * int -> int

- 'fun' is the keyword to define a function
- 'plus' is the function name that takes 2 arguments a and b of type int

- Function returns an evaluated expression 'a+b'
- Semicolon terminates the function

- Here we can see that plus is a function that takes 2 args
- ML definition says that all functions take single arg

- Function plus take 2 element tuple as arg
- To call plus, we place its name and tuple as arg

```
plus (3, 4);
==> val it = 7 : int
```

- Parenthesis is not part of function call syntax
- It delimits the tuple
- interpreter responds as follows:
- 'it' refers to understand variable
- On evaluation interpreter produces a value 7 of type 'int'

- Single arg function w/o parenthesiszing

  fun twice n:=int=n+n

  ==> val twice = fn:int-> int

  twice 2;

  ==> val it = 4:int

- We can add parenthesis in either declaration or call

  twice(2)

  ==> val it=4:int

Curried function in ML

```
fun curried_plus a = fn b : int => a + b;
==> val curried_plus = fn : int -> int -> int
```

- The type groups implicitly: int->(int->int)
- Maps an integer to a function that maps an integer to an integer

```
curried_plus 3;
==> val it = fn : int -> int
```

Another Notation

```
fun curried_plus a b : int = a + b;
==> val curried_plus = fn : int -> int -> int
```

- This is not a function with 2 arguments
- Function has single arg 'a'
- It returns a function that takes a single arg 'b'
- This second function returns a+b

Fold function using tuple notation

```
fun fold (f, i, l) =
case l of
nil => i
| h :: t => f (h, fold (f, i, t));
```

Curried version of fold

```
fun curried_fold f i l =
case l of
nil => i
| h :: t => f (h, curried_fold f i t);
```

```
curried_fold plus;
==> val it = fn : int -> int list -> int

curried_fold plus 0;
==> val it = fn : int list -> int

curried_fold plus 0 [1, 2, 3, 4, 5];
==> val it = 15 : int
```

ML vs Scheme

- ML notation is easier than scheme notation

```
curried_fold plus 0 [1, 2, 3, 4, 5]; (* ML *)

(((curried_fold +) 0) '(1 2 3 4 5)) ; Scheme
```