

# Functional Programming

How do we handle

- Sequences of actions
- Storing intermediate results
- Avoiding unnecessary computation

Example: Define a function "pfacts" that takes a list L of numbers as argument, and

1. Computes the sorted version of L
2. Computes the max and min of L
3. Prints L, the sorted version of L, max, and min.

# Functional Programming

How do we handle

- Sequences of actions
- Storing intermediate results
- Avoiding unnecessary computation

Example: Define a function "pfacts" that takes a list L of numbers as argument, and

1. Computes the sorted version of L
2. Computes the max and min of L
3. Prints L, the sorted version of L, max, and min.

---

```
(define SL) (define BIG) (define SMALL)
(define pfacts1 (lambda (L)
  (set! SL (isort L))
  (set! BIG (car SL))
  (set! SMALL (last1 SL))
  (write L) (newline) (write SL) (newline)
  (write BIG) (write-string " ")
  (write SMALL) #t ))
```

Imperative Style

Disadvantages:

- destructive assignment
- SL, BIG, SMALL, are global

# Functional Programming

How do we handle

- Sequences of actions
- Storing intermediate results
- Avoiding unnecessary computation

Example: Define a function "pfacts" that takes a list L of numbers as argument, and

1. Computes the sorted version of L
2. Computes the max and min of L
3. Prints L, the sorted version of L, max, and min.

---

```
(define pfacts2 (lambda (L)
  (let* ((SL (isort L))
        (BIG (car SL))
        (SMALL (lastel SL)) )
    (write L) (newline) (write SL)
    (newline) (write BIG)
    (write-string " " ) (write SMALL) #t) ))
```

Uses assignment, but neither destructive nor global:

SL, BIG, SMALL, are **local**

# Functional Programming

How do we handle

- Sequences of actions
- Storing intermediate results
- Avoiding unnecessary computation

Example: Define a function "pfacts" that takes a list L of numbers as argument, and

1. Computes the sorted version of L
2. Computes the max and min of L
3. Prints L, the sorted version of L, max, and min.

---

```
(define pfacts3 (lambda (L)
  (write L) (newline) (write (isort L))
  (newline) (write (car (isort L))))
  (write-string " ")
  (write (lastel (isort L))) #t))
```

## Functional Style

Disadvantage:

- sorted the list 3 times!

# Functional Programming

How do we handle

- Sequences of actions
- Storing intermediate results
- Avoiding unnecessary computation

Example: Define a function "pfacts" that takes a list L of numbers as argument, and

1. Computes the sorted version of L
2. Computes the max and min of L
3. Prints L, the sorted version of L, max, and min.

---

```
(define pfacts4 (lambda (L) (aux L (isort L))))  
(define aux (lambda (L SL)  
  (write L) (newline) (write SL) (newline)  
  (write (car SL)) (write-string " ")  
  (write (last1 SL)) #t))
```

Functional Style and Efficient

```
(define isort (lambda (L)
  (cond ((or (null? L) (null? (cdr L))) L)
        (#t (place (car L) (isort (cdr L)))))) )
```

```
(define place (lambda (e L)
  (cond ((null? L) (list e))
        ((>= e (car L)) (cons e L))
        (#t (cons (car L) (place e (cdr L))))))
```

```
(define lastel (lambda (L)
  (cond ((null? L) ())
        ((null? (cdr L)) (car L))
        (#t (lastel (cdr L))))))
```