

Control Operations

- Procedure Application

(procedure exp ...)

- Sequencing

(begin exp₁ exp₂ ...)

Control Operations (cont.)

- Conditionals

(if test consequent alternative)

(not exp)

(and exp ...)

(or exp ...)

(cond clause1 clause2 ...)

(case exp clause1 clause2 ...)

Local Binding

Can we do this?

```
(let ((sum (lambda (ls)
             (if (null? ls)
                 0
                 (+ (car ls) (sum (cdr ls)))))))
```

Local Binding

- Forms of **let**:

`(let ((var val) ...) exp ...)`

`(let* ((var val) ...) exp ...)`

`(letrec ((var val) ...) exp ...)`

`(let name ((var val) ...) exp ...)`

letrec

```
(letrec ((sum (lambda (ls)
  (if (null? ls)
      0
      (+ (car ls) (sum (cdr
ls)))))))
```

Named **let** example

```
(define fibonacci
  (lambda (n)
    (let fib ((i n))
      (cond
        ((= i 0) 0)
        ((= i 1) 1)
        (else (+
                (fib (- i 1))
                (fib (- i 2))))))))))
```

do Iteration

```
(do ((var val update) ...)
    (test res ...) exp ...)
```

```
(define factorial
  (lambda (n)
    (do ((i n (- i 1)) (a 1 (* a i)))
        ((zero? i) a))))
```

for-each mapping

(**for-each** *procedure list1 list2 ...*)

- similar to map except that:
 - does not return a list of results
 - application of procedure to elements of the lists occurs in order from left to right.

Predicates

`(eq? obj1 obj2)`

identical

`(equiv? obj1 obj2)`

equivalent

`(equal? obj1 obj2)`

same

List procedures

`(list obj ...)`

`(list? obj)`

`(length list)`

`(list-ref list n)`

`(list-tail list n)`

`(memq obj list)`

`(memv obj list)`

`(member obj list)`

Numbers

- Tons of procedures that operate on numbers:

`+ - * /`

`zero? positive negative? even?`

`modulo truncate round abs max min`

`sin cos tan sqrt exp log`

`number->string string->number`

Strings

`string=? string<? string>?`

`string<=? string>=?`

`string-ci=? string-ci<? string-ci>?`

`string-ci<=? string-ci>=?`

`make-string string-length`

`string-copy string-append substring`

`string->list list->string`

Vectors

- Vectors are basically lists with some optimization for referencing arbitrary elements in constant time.
- Vectors begin with `# (` and end with `)`
- Procedures:

`vector` `vector-length` `vector-ref`
`vector-set!`

`vector->list` `list->vector`