# Lisp Introduction



Sihui Li

CSCI-534, Colorado School of Mines



Spring 2022

# What is Lisp?

**Definition (Lisp)**

A family of programming languages based on s-expressions.



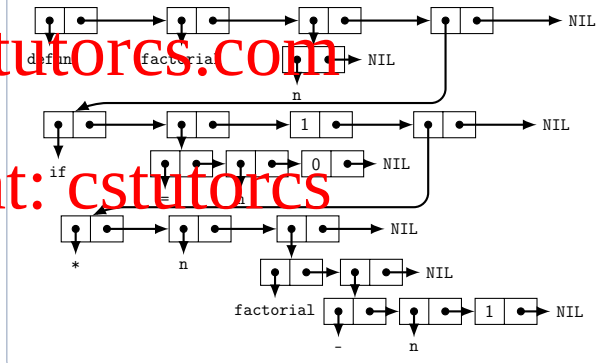
**"Math"**

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n \neq 0 \end{cases}$$

**Code**

```
(defun fact (n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

**Data Structure**



# Homoiconic

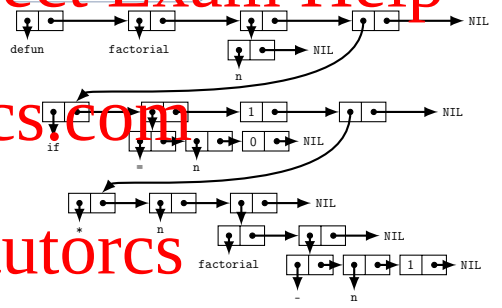
## Code is Data



Code

```
(defun fact (n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

Data Structure




*"data processing"* $\Longleftrightarrow$ *"code processing"*

# Introduction



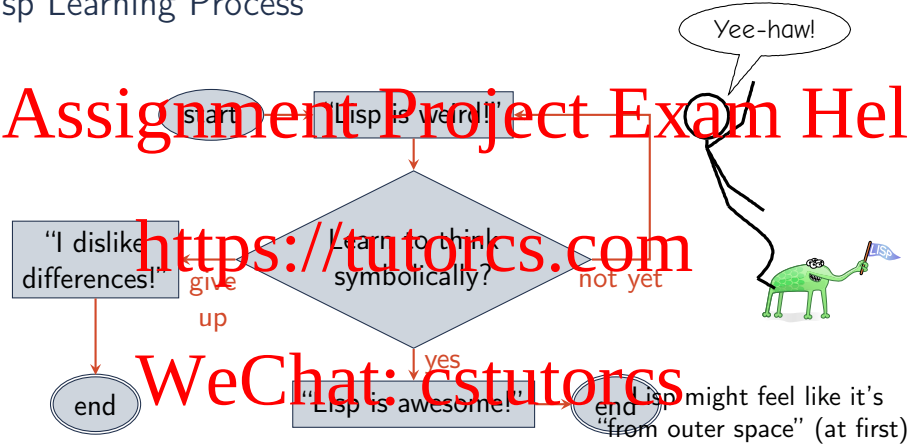
## Why study Lisp?

- Functional Programming:
  - Planning algorithms often are functional/recursive
  - Lisp has good support for functional programming.
- Symbolic Computing:
  - Planning algorithms must often process symbolic expressions
  - Lisp has good support for symbolic processing
- Understanding the abstractions in Lisp will make you a better programmer.

## Outcomes

- Understand Lisp abstractions
  - Symbols
  - S-expressions
  - First-class functions (closures)
- Implement Lisp programs in functional style
- (Review differential calculus and numerical methods)

# The Lisp Learning Process



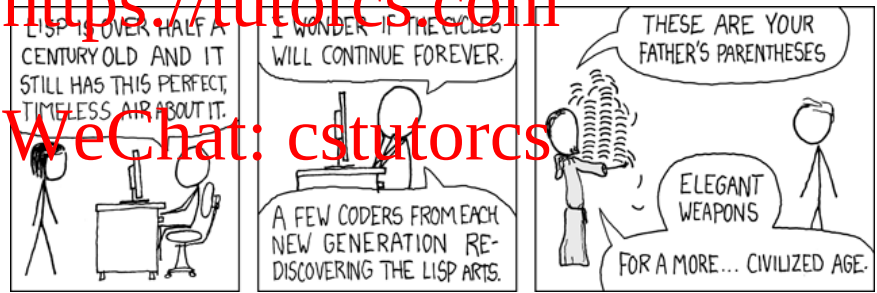



*The symbolic view of programming is* **different**, *often* **useful**.

# Lots of Irritating Silly Parenthesis?

"LISP has jokingly been described as 'the most intelligent way to misuse a computer'. I think that description a great compliment because it transmits the full flavour of liberation: it has assisted a number of our most gifted fellow humans in **thinking previously impossible thoughts**." [emphasis added]






https://xkcd.com/297/

# Outline

# Booleans and Equality

| "Math" | Lisp | Notes |
|---|---|---|
| False | `nil` | equivalent to the empty list `()` |
| True | `t` | or any non-`nil` value |
| $\neg a$ | `(not a)` | |
| $a = b$ | `(= a b)` | numerical comparison |
| $a = b$ | `(eq a b)` | same object |
| $a = b$ | `(eql a b)` | same object, same number and type, or same character |
| $a = b$ | `(equal a b)` | `eql` objects, or lists/arrays with equal elements |
| $a = b$ | `(equalp a b)` | = numbers, or same character (case-insensitive), or recursively-`equalp` cons cells, arrays, structures, hash tables |
| $a \neq b$ | `(not (= a b))` | similarly for other equality functions |

# Example: Lisp Equality Operators

- (= 1 1) ⇝ t
- (eq 1 1) ⇝ t
- (= 1 1.0) ⇝ t
- (eq 1 1.0) ⇝ nil
- (eql 1 1.0) ⇝ nil
- (equal 1 1.0) ⇝ nil
- (equalp 1 1.0) ⇝ t

- (= "a" "a") ⇝ error
- (eq "a" "a") ⇝ nil
- (eql "a" "a") ⇝ nil
- (equal "a" "a") ⇝ t
- (equal "a" "A") ⇝ nil
- (equalp "a" "A") ⇝ t
- (not t) ⇝ nil
- (not nil) ⇝ t
- (not "a") ⇝ nil

# Exercise: Lisp Equality Operators



- `(not 0)` ⇝ `nil`
- `(not 1)` ⇝ `nil`
- `(eq t (not nil))` ⇝ `t`
- `(eq t 1)` ⇝ `nil`
- `(eq nil (not 1))` ⇝ `t`
- `(eq nil (not "a"))` ⇝ `t`

- `(eq (list "a" "b") (list "a" "b"))` ⇝ `nil`
- `(equal (list "a" "b") (list "a" "b"))` ⇝ `t`
- `(eq (list "a" "b") (list "a" "B"))` ⇝ `nil`
- `(equal (list "a" "b") (list "a" "B"))` ⇝ `nil`
- `(equalp (list "a" "b") (list "a" "B"))` ⇝ `t`

## Inequality



| "Math" | Lisp |
|---|---|
| $a < b$ | `(< a b)` |
| $a \leq b$ | `(<= a b)` |
| $a > b$ | `(> a b)` |
| $a \geq b$ | `(>= a b)` |

# Function Definition



## DEFUN

Defines a new function in the global environment.

```
(defun FUNCTION-NAME ARGUMENTS BODY...)
```

## Pseudocode

**Procedure** increment($n$)

1 **return** $n + 1$;

## Common Lisp

function name function arguments

```
(defun increment (n)
  (+ n 1))
```

result

# Exercise: Function Definition

## Sine Cardinal



$$\text{sinc}\theta = \frac{\sin\theta}{\theta}$$

**Pseudocode**

**Procedure** sinc($\theta$)

1 **return** $\sin(\theta)/\theta$

**Common Lisp**

```
(defun sinc (theta)
  (/ (sin theta)
     theta))
```

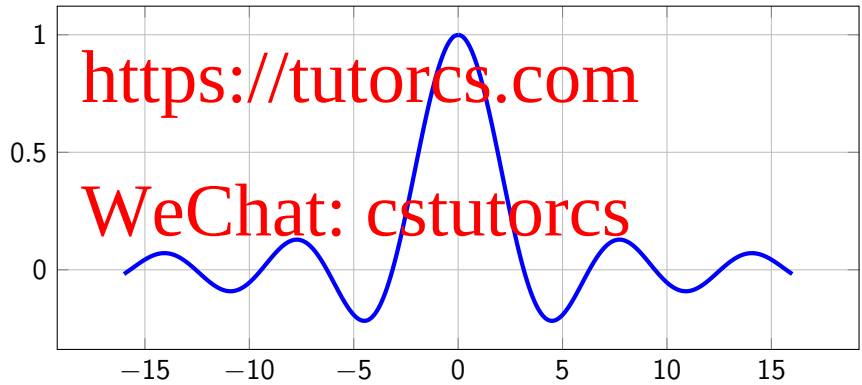


*What's wrong (mathematically) with this definition?*

# Limit of $\operatorname{sinc}\theta$

$$\lim_{\theta\to 0}\frac{\sin\theta}{\theta} \overset{\text{l'Hôpital's rule}}{\rightsquigarrow} \lim_{\theta\to 0}\frac{\frac{d}{d\theta}\sin\theta}{\frac{d}{d\theta}\theta} \rightsquigarrow \lim_{\theta\to 0}\frac{\cos\theta}{1} \mapsto 1$$





# Conditional

IF



## IF

Conditional execution based on a single test:

```
(if TEST THEN-EXPRESSION [ELSE-EXPRESSION])
```

### Pseudocode

**Procedure** even?(n)

```
1 if 0 = mod(n, 2) then
2 |   return true;
3 else
4 |   return false;
```

### Common Lisp

```
(defun even? (n)
  (if (= 0 (mod n 2))
      t
      nil))
```

test

then clause

else clause

# Exercise: Conditionals

IF



$$\text{sinc}(\theta) = \begin{cases} 1 & \text{if } \theta = 0 \\ \frac{\sin\theta}{\theta} & \text{if } \theta \neq 0 \end{cases}$$

**Pseudocode**

**Procedure** sinc($\theta$)

```
1 if 0 = θ then
2 |   return 1;
3 else
4 |   return sin(θ)/θ;
```

**Common Lisp**

```
(defun sinc (theta)
  (if (= 0 theta)
      1
      (/ (sin theta)
         theta)))
```

# Taylor Series

Represent function $f(x)$ as infinite sum of derivatives around point $a$:



$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a) + \frac{f'''(a)}{3!}(x-a) + \ldots$$



$$= \sum_{n=0}^{\infty} \left( \frac{f^{(n)}(a)}{n!}(x-a)^n \right)$$

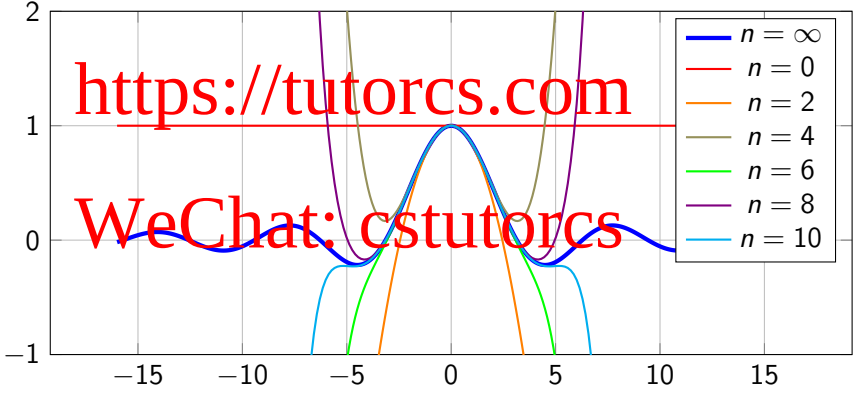


*Polynomial approximation of functions*

# Sinc Taylor Series

$$\frac{\sin\theta}{\theta} = 1 - \frac{\theta^2}{6} + \frac{\theta^4}{120} - \frac{\theta^6}{5040} + \frac{\theta^8}{362880} - \frac{\theta^{10}}{39916800} + \cdots$$





# Conditional

## COND

**COND**

Conditional execution of the first clause with a true test: `(cond CLAUSES...)`

where each clause is of the form: `(TEST EXPRESSIONS...)`



### Pseudocode

**Procedure** sign(n)

1 **if** $n > 0$ **then return** $1$ ;
2 **else if** $n < 0$ **then return** $-1$ ;
3 **else return** $0$ ;

### Common Lisp

```
(defun sign (n)
  (cond ((> n 0) 1)
        ((< n 0) -1)
        (t 0)))
```

# Exercise: Conditionals

COND

$$\frac{\sin\theta}{\theta} = 1 - \frac{\theta^2}{6} + \frac{\theta^4}{120} + \ldots$$



**Pseudocode**

**Procedure** sinc($\theta$)

1 **if** $0 = \theta$ **then**
2 | **return** 1;
3 **else if** $\theta^2 < .00001$ **then**
4 | **return** $1 - \theta^2/6 + \theta^4/120$;
5 **else**
6 | **return** $\sin(\theta)/\theta$;



**Common Lisp**

```
(defun sinc (theta)
  (cond
    ((= 0 theta) 1)
    ((< (* theta theta) .00001)
     (+ 1
        (/ (expt theta 2) -6)
        (/ (expt theta 4) 120)))
    (t (/ (sin theta)
          theta))))
```

# Recursion



## Example: Factorial



$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n \neq 0 \end{cases}$$

Recursion: A function (or other object) defined in terms of itself

Base Case: Terminating condition

Recursive Case: Reduction towards the base case



**Procedure** factorial($n$)

```
1 if 0 = n then // base case
2     return 1;
3 else // recursive case
4     return n * factorial(n − 1);
```

# Example: Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n \neq 0 \end{cases}$$



**Pseudocode**

**Procedure** factorial($n$)

```
1 if 0 = n then // base case
2 |   return 1;
3 else // recursive case
4 |   return
  |     n * factorial(n − 1);
```

**Common Lisp**

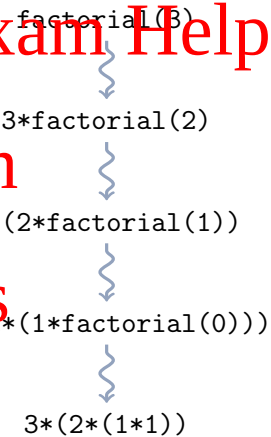
```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

# Factorial Execution Trace



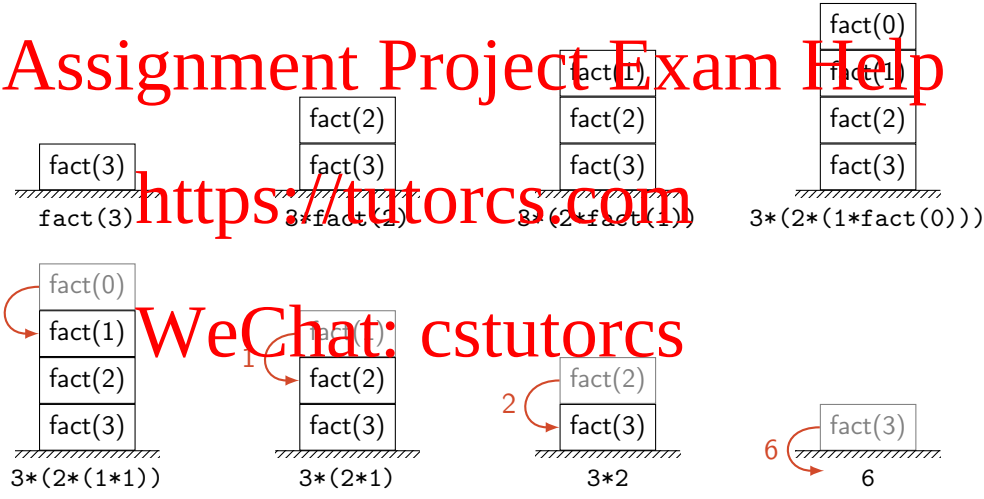
**Procedure** factorial($n$)

```
1 if 0 = n then // base case
2 |   return 1;
3 else // recursive case
4 |   return n * factorial(n − 1);
```



# Factorial Call Stack





## Exercise: Recursion, Fibonacci Sequence



$$(1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55, \ldots)$$



$$\mathrm{fib}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{if } n \geq 2 \end{cases}$$

# Exercise: Recursion, Fibonacci Sequence

continued



**Pseudocode**

**Procedure** fib($x$)

```
1 if 0 = n then return 1 ;
2 else if 1 = n then return 1 ;
3 else
4     return fib(n − 1) + fib(n − 2);
```



**Common Lisp**

```
(defun fib (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
```

# Exercise: Recursion, Accumulate



## Iterative

**Function** accumulate(S)

```
1 a ← 0;
2 i ← 0;
3 while i < |S| do
4     a ← a + S_i;
5 return a;
```

## Recursive

**Function** accumulate(S)

```
1 if S then // Recursive Case
2     return
        car(S) + accumulate(cdr(S));
3 else // Base Case
4     return 0;
```

# Exercise: Recursion, Accumulate

## Lisp Code



**Function** accumulate(S)

1 **if** $S$ **then** // Recursive Case
2 **return** $\mathbf{car}(S) + \text{accumulate}(\mathbf{cdr}(S))$;
3 **else** // Base Case
4 **return** 0;





Recursive Implementation of Accumulate

```
(defun accumulate (list)
  (if list
      ;; recursive case
      (+ (car list)
         (accumulate (cdr list)))
      ;; base case
      0))
```

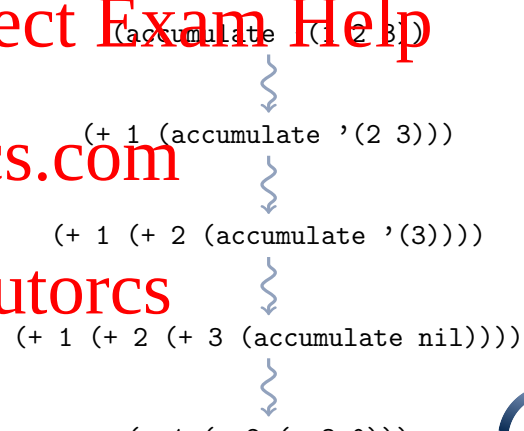
# Exercise: Recursion, Accumulate

Execution Trace



**Recursive Implementation of Accumulate**

```
(defun accumulate (list)
  (if list
      ;; recursive case
      (+ (car list)
         (accumulate (cdr list)))
      ;; base case
      0))
```



# First-class functions



**Definition: First-class functions**

A programming language has **first-class functions** when it treats functions like any other variable or object. First-class functions can be:

- Assigned to variables
- Passed as arguments to other functions
- Returned as the result of other functions

# Local Variables

## LET, LET*

LET and LET* create and initialize new local variables. LET operates in "parallel" and LET* operates sequentially.



### Example (LET)

```
(let ((a 1))
  (let ((a 2)
        (b a))
    (print (list a b))))
```

Output

```
(2 1)
```

### Example (LET*)

```
(let ((a 1))
  (let* ((a 2)
         (b a))
    (print (list a b))))
```

Output

```
(2 2)
```

# Closure

**Definition (Closure)**

A function and an associated set of variable definitions. From "closed expression"



## C Function Pointer

```c
/* Definition */
struct context {
    int val;
};

int adder(struct context *cx, int x) {
    return cx->val + x;
}

/* Usage */
struct context c;
c.val = 1;
int y = adder(c,2);
```



## Java Class

```java
// Definition
class Adder {
    public int a;
    public Adder(int a_) {
        a = a_;
    }
    public int call(int x) {
        return x+a;
    }
}

// Usage
Adder A = new Adder(1);
int y = A.call(2);
```

# Closures in Lisp: Local Functions

**LABELS**

Defines local functions and executes body using those local functions:

```
(labels ((FUNCTION-NAME VARIABLES FUNCTION-BODY)...)  LABELS-BODY)
```



**Example**

```
(let ((a 1))
  (labels ((adder (x)
             (+ x a)))
    (adder 2)))
```

**Output**

3

# Closures in Lisp: Anonymous Functions

**LAMBDA**

Defines an anonymous function:

```
(lambda VARIABLES FUNCTION-BODY)
```

**FUNCALL**

Apply a function to the provided arguments:

```
(funcall FUNCTION ARGUMENTS...)
```



**Example**

```
(let ((a 1))
  (funcall (lambda (x)
             (+ x a))
           2))
```

**Output**

3

# Value and Function Namespaces



## Value Namespace

- Records values
- Local: `let`, `let*`
- Global: `defparameter`

## Function Namespace

- Records function definitions
- Local: `labels`, `flet`
- Global: `defun`

## Example

```
(defun foo (x) (+ 1 x))

(let ((foo 10))
  (print foo)          ; => 10
  (print (foo 1))      ; => 2
  (print (foo foo)))   ; => 11
```

### Output

```
10
2
11
```

# function and funcall

## FUNCTION

Returns the functional value of a name:

(function NAME)

⇝ The function bound to name

### Example

- (function +)
- (#' +)
- (defun foo (x) (+ 1 x))
  #'foo
- (labels ((foo (x) (+ 1 x)))
    #'foo)

## FUNCALL

Apply a function to the provided arguments:

(funcall FUNCTION ARGS...)

⇝ Return value of FUNCTION called on ARGS....

### Example

- (funcall (lambda (x)
    (+ 1 x))
  1)

  ⇝ 2
- (funcall #'+ 1 2) ⇝ 3

# Example Domain: Numerical Integration

## Runge-Kutta Methods



Given:
- Derivative: $\frac{d}{dt}x(t) = f(x, t)$
- Initial time: $t_0$
- Final time: $t_n$
- Initial value: $x(t_0)$



Find: $x(t_n)$

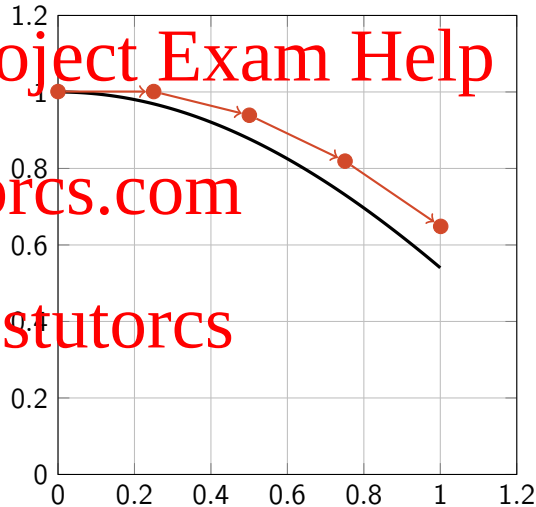


Solution: Follow derivative along discrete time intervals $\Delta t$ from $t_0$ to $t_n$

# Example: Runge-Kutta 1 (Euler's Method)



$$x_{i+1} \approx x_i + \Delta t * f(x_i, t_i)$$



# Example: Runge-Kutta 1 (Euler's Method)

continued



$$x_{i+1} \approx x_i + \Delta t * f(x_i, t_i)$$

**Procedure** euler-step($dx, dt, x_0$)

1 **return** $x_0 + dx * dt$;

```
(defun euler-step (dx dt x0)
  (+ x0
     (* dx dt)))
```

# Example: Runge-Kutta 2 (Midpoint Method)



$$x_{i+1} \approx x_i + \Delta t * f(x_i + \frac{\Delta t}{2} f(x_i, t_i),\ t + \frac{\Delta t}{2}) \quad \overbrace{\approx \dot{x}(t_i + \frac{\Delta t}{2})}$$



**Procedure** rk2-mid($f, t_0, x_0, dt$)

```
1 function ks(c, k) is
2     let
3         k_t ← c * dt;
4         x ← euler-step(k, k_t, x_0)
5     in return f(x, t_0 + k_t);
6 let
7     k_0 ← f(x_0, t_0);
8     k_1 ← ks(1/2, k_0);
9 in return x_0 + dt * k_1;
```

```
(defun rk2-mid-step (f t0 x0 dt)
  (labels
      ((ks (c k)
         (let* ((kt (* c dt))
                (x (euler-step k kt x0)))
           (funcall f x (+ t0 kt)))))
    (let* ((k0 (funcall f x0 t0))
           (k1 (ks (/ 1 2) k0)))
      (+ x0 (* dt k1)))))
```



## Exercise: Runge-Kutta 2 (Heun's Method)



$$x_{i+1} \approx x_i + \frac{\Delta t}{2} * \overbrace{f(x_i, t_i)}^{\dot{x}(t_i)} + \frac{\Delta t}{2} * \overbrace{f(x_i + (\Delta t) f(x_i, t_i), t + \Delta t)}^{\approx \dot{x}(t+\Delta t)}$$

$$\approx x_i + \frac{\Delta t}{2} k_0 + \frac{\Delta t}{2} k_1$$

*Averaging derivatives at current and next point.*

# Exercise: Runge-Kutta 2 (Heun's Method)

continued

**Procedure** heun$(f, t_0, x_0, dt)$

```
1 function ks(c, k) is
2     let
3         k_t ← c * dt;
4         x ← euler-step(k, k_t, x_0);
5     in return f(x, t_0 + k_t);
6 let
7     k_0 ← f(x_0, t_0);
8     k_1 ← ks(1, k_0);
9 in return x_0 + dt/2 * (k_0 + k_1);
```

```
(defun rk2-heun (f t0 x0 dt)
  (labels
      ((ks (c k)
         (let* ((kt (* c dt))
                (x (euler-step k kt x0)))
           (funcall f x (+ t0 kt)))))
    (let* ((k0 (funcall f x0 t0))
           (k1 (ks 1 k0)))
      (+ x0
         (* (/ dt 2)
            (+ k0 k1))))))
```

# Exercise: Runge-Kutta 4



$$x_{i+1} \approx x_i + \frac{\Delta t}{6} k_0 + \frac{\Delta t}{3} k_1 + \frac{\Delta t}{3} k_2 + \frac{\Delta t}{6} k_3$$

$$\approx x_i + \frac{\Delta t}{6}(k_0 + k_3) + \frac{\Delta t}{3}(k_1 + k_2)$$



where:

- $k_0 = f(x_i, t_i)$ (current)
- $k_1 = f(x_i + \frac{\Delta t}{2} k_0,\ t_i + \frac{\Delta t}{2})$ (midpoint)
- $k_2 = f(x_i + \frac{\Delta t}{2} k_1,\ t_i + \frac{\Delta t}{2})$ (midpoint)
- $k_3 = f(x_i + (\Delta t) k_2,\ t_i + \Delta t)$ (next)



*Weighted average at current point, midpoint, and next point.*

# Exercise: Runge-Kutta 4

continued

**Procedure** $rk4\_step(f, t_0, x_0, dt)$

```
function ks(c, k) is
    let
        k_t ← c * dt;
        x ← euler-step(x, k_t, x0);
    in return f(x, t0 + k_t);
let
    k0 ← f(x0, t0);
    k1 ← ks(1/2, k0);
    k2 ← ks(1/2, k1);
    k3 ← ks(1, k2);
in return x0 + dt/6 * (k0 + k2) +
   dt/3 * (k0 + k1);
```

```
(defun rk4-step (f t0 x0 dt)
  (labels
    ((ks (c k)
       (let* ((kt (* c dt))
              (x (euler-step k kt x0)))
         (funcall f x (+ t0 kt)))))
    (let* ((k0 (funcall f x0 t0))
           (k1 (ks (/ 1 2) k0))
           (k2 (ks (/ 1 2) k1))
           (k3 (ks 1 k2)))
      (+ x0
         (* (/ dt 6) (+ k0 k3))
         (* (/ dt 3) (+ k1 k2)))))))
```

# Euler (RK-1) Integration

**Procedure** int-rk1$(f, t_0, t_n, dt, x_0)$

1 **if** $t_0 \geq t_n$ **then** // Base Case
2 **return** $x_0$;
3 **else** // Recursive Case
4 **let**
5 $dx \leftarrow f(x_0, t_0)$;
6 $x \leftarrow$ euler-step$(dx, dt, x_0)$;
7 $t_1 \leftarrow t_0 + dt$;
8 **in return** int-rk1$(f, t_1, t_n, dt, x)$;

```
(defun int-rk1 (f t0 tn dt x0)
  (if (>= t0 tn)
      x0
      (let* ((dx (funcall f x0 t0))
             (x (euler-step dx dt x0))
             (t1 (+ t0 dt)
             (int-rk1 f t1 tn dt x))))))
```

# RK-2 Integration

**Procedure** int-rk2($f, t_0, t_n, dt, x_0$)

1 **if** $t_0 \geq t_n$ **then** // Base Case
2 | **return** $x_0$;
3 **else** // Recursive Case
4 | **let**
5 | | $x \leftarrow$ rk2-heun$(f, t_0, x_0, dt)$;
6 | | $t_1 \leftarrow t_0 + dt$
7 | **in return** int-rk2$(f, t_1, t_n, dt, x)$;

```
(defun int-rk2 (f t0 tn dt x0)
  (if (>= t0 tn)
      x0
      (let ((x (rk2-heun f t0 x0 dt))
            (t1 (+ t0 dt)))
        (int-rk2 f t1 tn dt x))))
```

# Exercise: Multi-method RK Integration



**Procedure** int-rkx$(s, f, t_0, t_n, dt, x_0)$

1 **if** $t_0 \geq t_n$ **then** // Base Case
2 | **return** $x_0$;
3 **else** // Recursive Case
4 | **let**
5 | | $x \leftarrow s(f, t_0, x_0, dt)$;
6 | | $t_1 \leftarrow t_0 + dt$;
7 | **in return**
| | int-rkx$(s, f, t_1, t_n, dt, x)$;

```
(defun int-rkx (s f t0 tn dt x0)
  (if (>= t0 tn)
      x0
      (let ((x (funcall s f t0 x0 dt))
            (t1 (+ t0 dt)))
        (int-rkx s f t1 tn dt x))))
```

# Higher-order functions

**Definition: Higher-order function**

A function that takes another function as an argument or returns another function as its result.



**Example (Passing)**

```
Function f(g,a)
1 return g(42, a);
```

**Example (Returning)**

```
Function f(a)
1 Function g(b) is
2     return a + b;
3 return g;
```

**Counterexample**

```
Function f(a,b)
1 return a + b;
```

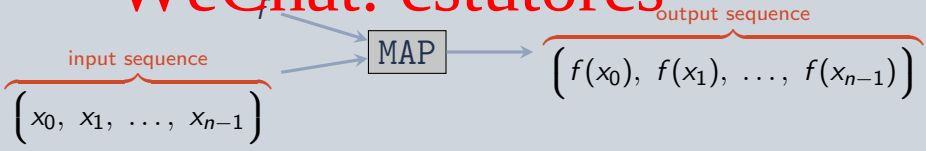
# Map function

**Definition (map)**

Apply a function to every member of an input sequence, and collect the results into the output sequence.

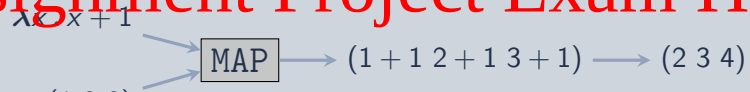
$$\texttt{map} \quad : \quad \underbrace{(\mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \quad \times \quad \underbrace{\mathbb{X}^n}_{\text{input sequence}} \quad \mapsto \quad \underbrace{\mathbb{Y}^n}_{\text{output sequence}}$$

**Illustration**



# Example: Map

+1






¬

# Algorithm: Map function



**Functional Map**

**Procedure** map(f,s)

```
1 if empty(s) then /* s is empty */
2 |   return NIL
3 else /* s has members */
4 |   return cons(f(first(s)), map(f, rest(s)));
```

**Imperative Map**

**Procedure** map(f,s)

```
1 n ← length(s);
2 Y ← make-sequence(n);
3 i ← 0;
4 while i < n do
5 |   Y[i] ← f(s[i]);
6 |   i ← i + 1;
7 return Y;
```

# Example: Map

## Example (Illustration)


## Example (Lisp)

```
(map 'list                ; result type
     (lambda (x) (+ 1 x)) ; function
     (list 1 2 3))        ; sequence
;; RESULT: (2 3 4)
```

# Fold-left

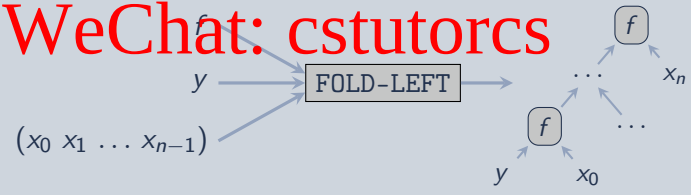
**Definition (fold-left)**

Apply a binary function to each member of a sequence and the prior result, starting from the left.

$$\text{fold-left} : \underbrace{(\mathbb{Y} \times \mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

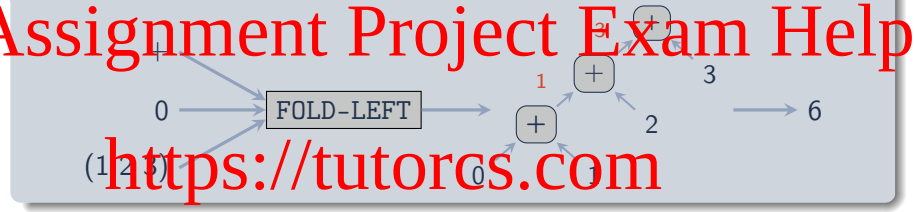


**Illustration**

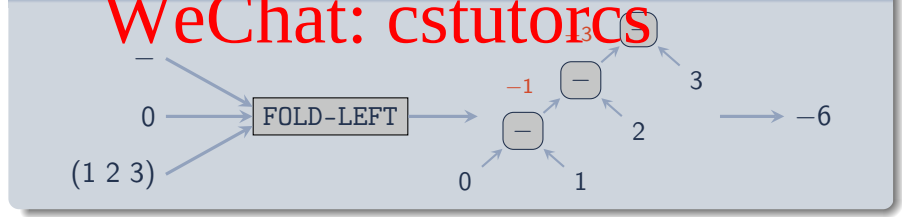


# Example: Fold-left

## Example (Addition)




## Example (Subtraction)



# Algorithm: Fold-left



## Imperative

**Function** fold-left(f, y, X)

```
1 i ← 0;
2 while i < |X| do
3 |   y ← f(y, X_i);
4 |   i ← i + 1;
5 return y;
```

## Functional

**Function** fold-left(f, y, X)

```
1 if empty(X) then return y;
2 else
3 |   let y' ← f(y, first(X)) in
4 |   return fold-left(f, y', rest(X));
```

# Example: Fold-left in Lisp



**Example (Addition)**

```
(reduce #'+
        '(1 2 3)
        :initial-value 0)
;;; => (+ (+ (+ 0 1) 2) 3)
;;; => 6
```



**Example (Subtraction)**

```
(reduce #'-
        '(1 2 3)
        :initial-value 0)
;;; => (- (- (- 0 1) 2) 3)
;;; => -6
```



# Exercise: Fold-Left Reverse



$$(a_0\ a_1\ \ldots\ a_{n-1}\ a_n) \overset{\text{reverse}}{\rightsquigarrow} (a_n\ a_{n-1}\ \ldots\ a_1\ a_0)$$

**Procedure** reverse(*L*)

```
1 function h(r, x) is
2     return cons(x, r) ;
3 return fold-left(h, NIL, L)
```

```
(defun reverse-fold (list)
  (reduce (lambda (reversed x)
            (cons x reversed))
          list
          :initial-value nil))
```

# Fold-right

**Definition (fold-right)**

Apply a binary function to each member of a sequence and the prior result, starting from the right.

$$\text{fold-right} : \underbrace{(\mathbb{X} \times \mathbb{Y} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

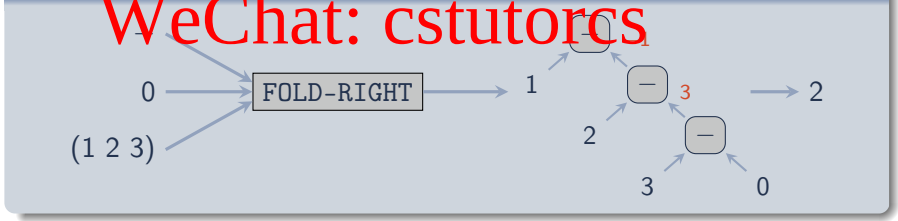


**Illustration**

# Example: Fold-right

## Example (Addition)


## Example (Subtraction)



# Algorithm: Fold-right



## Procedural

**Function** fold-right(f,y,X)

1 $i \leftarrow |X| - 1$;
2 **while** $i \geq 0$ **do**
3 $\quad y \leftarrow f(X_i, y)$ ;
4 $\quad i \leftarrow i - 1$ ;
5 **return** $y$;

## Recursive

**Function** fold-right(f,y,X)

1 **if** `empty(X)` **then return** $y$;
2 **else**
3 $\quad$ **let** $y' \leftarrow$ `fold-right(f, y, rest(X))` **in**
4 $\quad\quad$ **return** `f(first(X), y')`;

# Example: Fold-right in Lisp



## Example (Addition)

```
(reduce #'+
        '(1 2 3)
        :initial-value 0
        :from-end t)
;;; => (+ 1 (+ 2 (+ 3 0)))
;;; => 6
```

## Example (Subtraction)

```
(reduce #'-
        '(1 2 3)
        :initial-value 0
        :from-end t)
;;; => (- 1 (- 2 (- 3 0)))
;;; => 2
```

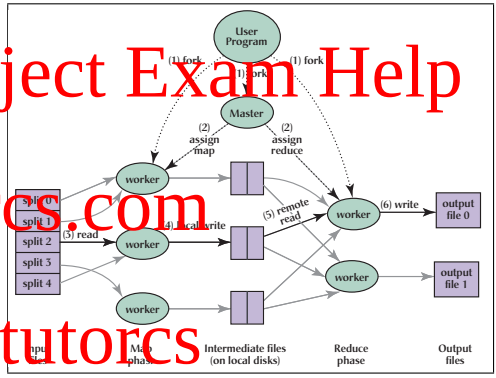
# Application: MapReduce

**Function** MapReduce(f,g,X)

```
1 Y ← parallel-map(f, X);
2 return reduce(g, Y);
```



- Idea:
  - (parallel) map
  - (serial) reduce/fold
- Provides scalability, fault-tolerance
- Implementations:
  - Google MapReduce
  - Apache Hadoop




Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM. 2008.

# Outline

# Data Types



**Definition (Data Type)**

A classification of data/objects based on how the data/object is intended to or able to be use.

The set of values a variable may take.

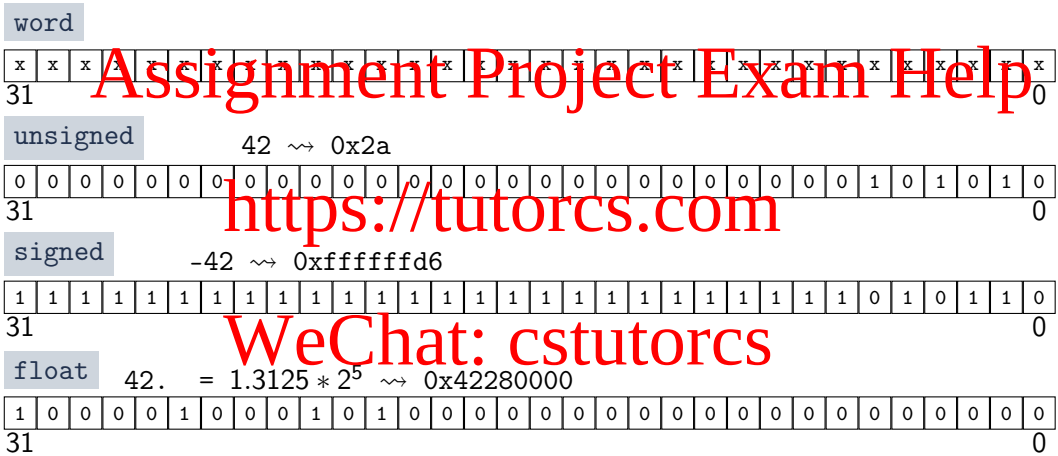
**Example**

- `int`
- `float`
- List
- String
- Structures:
  - $\texttt{int} \times \texttt{string}$
  - $\texttt{float}^4$
- Function:
  - $\texttt{int} \times \texttt{int} \mapsto \texttt{bool}$

# Data Type Systems

- Type Checking/Binding
  - Static: Check types at compile time (statically)
  - Dynamic: Check types at run time (dynamically).
- Type Enforcement
  - Strong: Object types are strictly enforced
  - Weak: Objects can be treated as different types (casting, "type punning")
- Examples:
  - C: static/weak
  - Python: dynamic/strong
  - ML and Haskell: static/strong
  - Lisp: dynamic (mostly) / strong

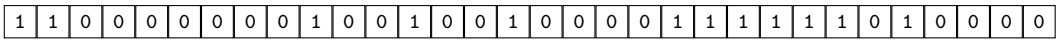
# Machine Words – Representing Data



# Words and Types



word






0xc0490fd0 $\overset{?}{\rightsquigarrow}$ −1068953648 (signed)

0xc0490fd0 $\overset{?}{\rightsquigarrow}$ 3226013648 (unsigned)

0xc0490fd0 $\overset{?}{\rightsquigarrow}$ −3.141590 (float)

0xc0490fd0 $\overset{?}{\rightsquigarrow}$ valid pointer

# Type Tags




## SBCL Tags (32-bit)

| Type | Tag |
|---|---|
| Even Fixnum | 000b |
| Odd Fixnum | 100b |
| Instance Pointer | 001b |
| List Pointer | 011b |
| Function Pointer | 101b |

data tag

0x180921FA × 000b ⇝ even fixnum

(0x180921FA >> 2) ⇝ 806503412

# Example: Tagged Storage

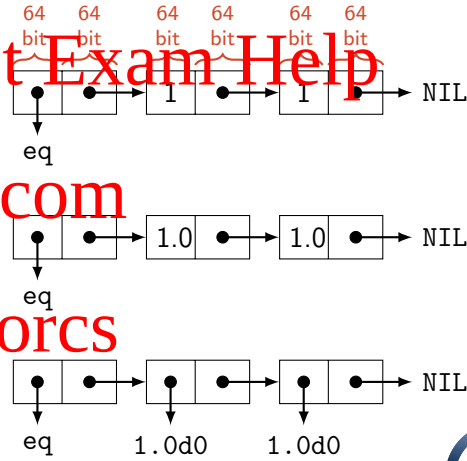


64-bit SBCL:

Fixnum: `(eq 1 1)` ⇝ `t`

Single Float: `(eq 1.0s0 1.0s0)` ⇝ `t`

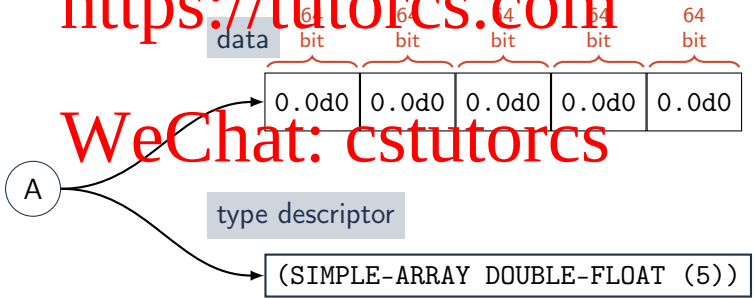
Double Float: `(eq 1.0d0 1.0d0)` ⇝ `nil`



# Example: SBCL Arrays

```
(let ((a (make-array 5
                     :element-type 'double-float)))
  ;; ....
  )
```





## Manual Memory Management



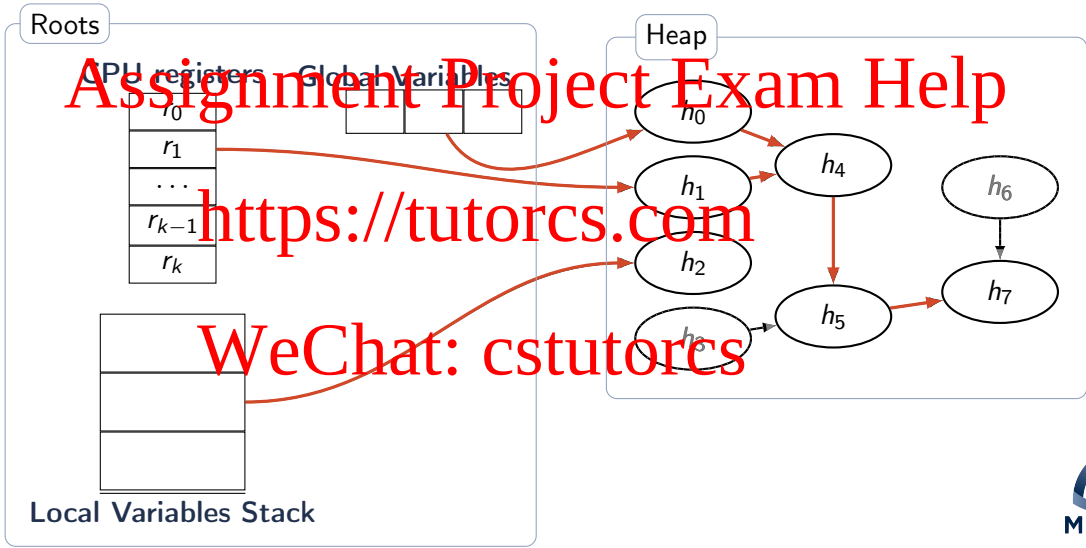
**`malloc(n)`**

1. Find a free block of at least $n$ bytes
2. If no such block, get more memory from the OS
3. Return pointer to the block

**`free(ptr)`**

1. Add block back to the free list(s)

# Garbage Collection



## Outline



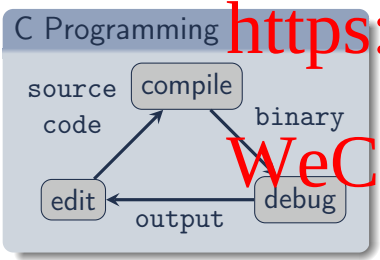


# Lisp Programming Environment

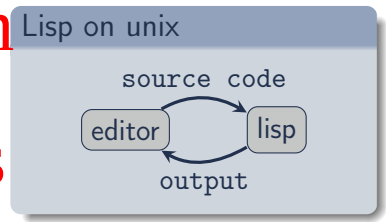


- Lisp Editor, Compiler, Debugger available at runtime
- Interactive development: Read-Eval-Print Loop (REPL)



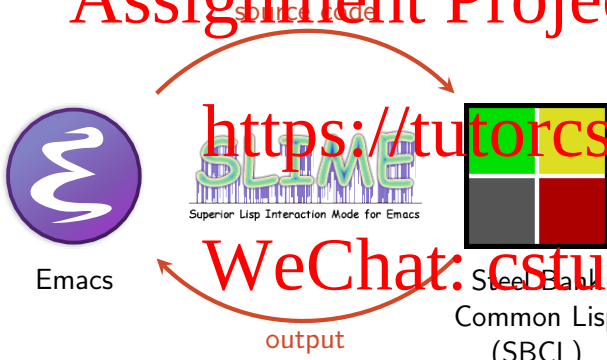






# SLIME Demo






- SLIME, `pstree`
- Read-Eval-Print-Loop (REPL)
- `DEFUN`
- `DISASSEMBLE`
- Re-`DEFUN`

## SLIME Basics



- `C`: control
- `M`: Meta / Alt
- Frequently used:
  - C-c C-k Compile and load file
  - C-x C-e Evaluate expression before the point
  - C-M-x Evaluate `defun` surround the point
- `Tab`: auto-indent line/region
- See `SLIME` drop-down in menu bar for more
- `https://common-lisp.net/project/slime/doc/html/`

# Style Notes

## New Lines

Newlines emphasize structure:

### Good Style

```
(and (or a (not b))
     (or b c))
```

### Bad Style

```
(and (or a (not b)) (or b c))
```

## Closing Parenthesis

Closing parenthesis on same line:

### Good Style

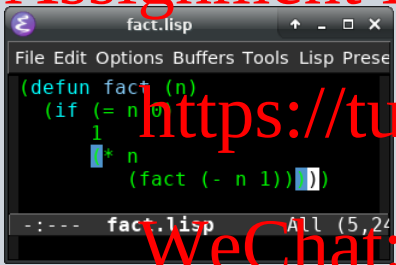
```
(defun foo (x)
  (+ 1 x))
```

### Bad Style

```
(defun foo (x)
  (+ 1 x)
)
```

# Emacs Tips

## Parenthesis Matching




.emacs

```
(show-paren-mode 1)
```

## Viper Mode

- vi implementation/emulator in Emacs
- User Manual

.emacs

```
(setq viper-mode t)
(require 'viper)
```

# Why use Lisp?

(Why learn something different?)



- **Functional Programming:**
  - Planning algorithms often are functional/recursive
  - Lisp has good support for functional programming.
- **Symbolic Computing:**
  - Planning algorithms must often process symbolic expressions
  - Lisp has good support for symbolic processing
- A good fit for (the first half of) this course

# References


- Peter Seibel. *Practical Common Lisp*. http://www.gigamonkeys.com/book/
- Common Lisp Hyperspec. http://www.lispworks.com/documentation/HyperSpec/Front/index.htm
- Paul Graham. *ANSI Common Lisp*.