# Lisp Introduction

Dr. Neil T. Dantam

CSCI-498/598 RPM, Colorado School of Mines

Spring 2018

# Outline

# Outline

# What is Lisp?

**Definition: Lisp**

A family of programming languages that are based on s-expressions.

# Major Lisp Dialects

**Scheme**

- IEEE Standard
- Simple and clean

**Common Lisp**

- ANSI Standard
- Featureful
- Good compilers
- Efficient C interop.

**Clojure**

- JVM-based
- Good Java interop.
- CLR and Javascript also
- Concurrency features

# 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)` $\leadsto$ `t`
- `(eq 1 1)` $\leadsto$ `t`
- $(= \overbrace{1}^{\text{integer}}\ \overbrace{1.0}^{\text{float}})$ $\leadsto$ `t`
- $(\text{eq}\ \overbrace{1}^{\text{integer}}\ \overbrace{1.0}^{\text{float}})$ $\leadsto$ `nil`
- $(\text{eql}\ \overbrace{1}^{\text{integer}}\ \overbrace{1.0}^{\text{float}})$ $\leadsto$ `nil`
- $(\text{equal}\ \overbrace{1}^{\text{integer}}\ \overbrace{1.0}^{\text{float}})$ $\leadsto$ `nil`
- $(\text{equalp}\ \overbrace{1}^{\text{integer}}\ \overbrace{1.0}^{\text{float}})$ $\leadsto$ `t`

- `(= "a" "a")` $\leadsto$ **error**
- `(eq "a" "a")` $\leadsto$ `nil`
- `(eql "a" "a")` $\leadsto$ `nil`
- `(equal "a" "a")` $\leadsto$ `t`
- `(equal "a" "A")` $\leadsto$ `nil`
- `(equalp "a" "A")` $\leadsto$ `t`
- `(not t)` $\leadsto$ `nil`
- `(not nil)` $\leadsto$ `t`
- `(not "a")` $\leadsto$ `nil`

# Exercise: Lisp Equality Operators

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

- `(eq t 1)` ⇝
- `(eq nil (not 1))` ⇝

- `(eq nil (not "a"))` ⇝

- `(eq (list "a" "b") (list "a" "b"))` ⇝
- `(equal (list "a" "b") (list "a" "b"))` ⇝

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

- `(equalp (list "a" "b") (list "a" "B"))` ⇝

# Inequality

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

# Function Definition

**Procedure** increment(n)

**return** $n + 1$;

$$(\textbf{defun}\ \overbrace{\texttt{increment}}^{\text{function name}}\ \overbrace{(n)}^{\text{function arguments}}\ \underbrace{(+\ n\ 1)}_{\text{result}})$$

# Exercise: Function Definition

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

## Pseudocode

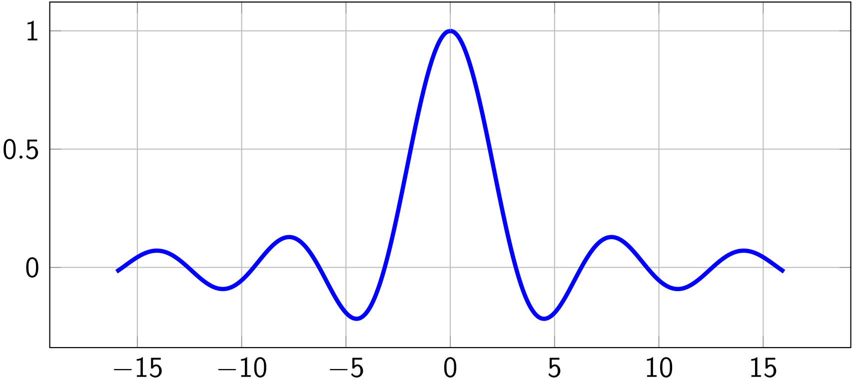
**Procedure** sinc($\theta$)

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

## Common Lisp

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

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



# Conditional

## IF

**Procedure** even?(n)

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

$$(\textbf{defun}\ \mathtt{even?}\ (n)\ \ (\textbf{if}\ \overbrace{(=0\ (\mathtt{mod}\ n\ 2))}^{\text{test}}\ \ \overbrace{\textbf{t}}^{\text{then clause}}\ \ \underbrace{\textbf{nil}}_{\text{else clause}}))$$

# Exercise: Conditionals

IF

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

## Pseudocode

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

```
if 0 = θ then
    return 1;
else
    return sin(θ)/θ;
```

## Common Lisp

# Taylor Series

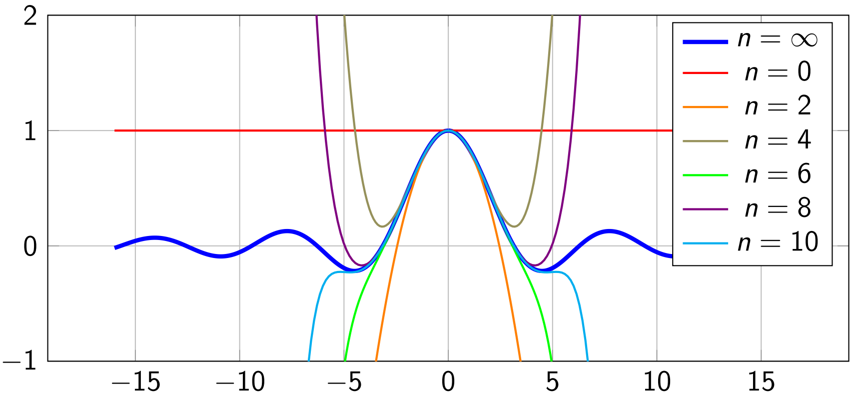
Represent function $f(x)$ as infinite sum of derivatives around point $a$:

$$\begin{aligned} f(x) &= f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a) + \frac{f'''(a)}{3!}(x-a) + \ldots \\ &= \sum_{n=0}^{\infty} \left( \frac{f^{(n)}(a)}{n!}(x-a)^n \right) \end{aligned}$$

*Polynomial approximation of functions*

# Sinc Taylor Series

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



# Conditional

## COND

**Procedure** sign(n)

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

$$(\textbf{defun}\ \texttt{sign}\ (n)$$

$$(\textbf{cond}\quad (\overbrace{(> n\ 0)}^{\text{test}}\ \overbrace{1}^{\text{result}})$$

$$(\overbrace{(< n\ 0)}^{\text{test}}\ \overbrace{-1}^{\text{result}})$$

$$(\overbrace{\mathbf{t}}^{\text{test}}\ \overbrace{0}^{\text{result}})))$$

# Exercise: Conditionals

COND

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

## Pseudocode

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

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

## Common Lisp

# Example: Factorial

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

## Pseudocode

**Procedure** factorial($x$)

**if** $0 = x$ **then**
    **return** $1$;
**else**
    **return** $x * \texttt{factorial}(x - 1)$;

## Common Lisp

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

# Exercise: 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: Fibonacci Sequence

continued

**Pseudocode**

**Common Lisp**

# Numerical Integration

## Runge-Kutta Methods

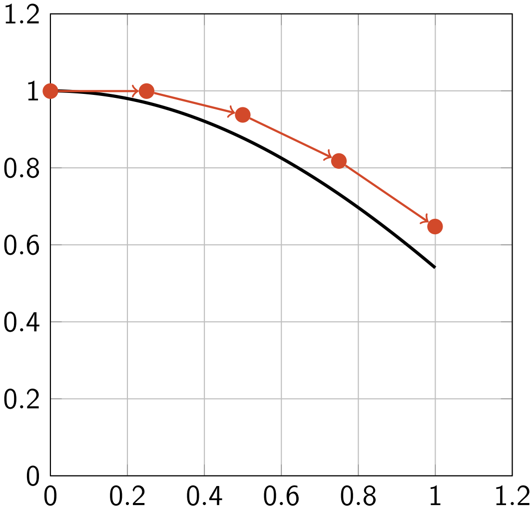
Given:
- Derivative: $\frac{d}{dt}x(t) = f(x,t)$
- Initial time: $t_0$
- Final time: $t_n$
- Initial value: $x(t_0)$

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)

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

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

**function** $ks(c, k)$ **is**
- $x \leftarrow$ `euler-step`$(k, c * dt, x_0)$;
- **return** $f(x,\ t + c * dt)$;

$k_0 \leftarrow f(x_0, t_0)$;
$k_1 \leftarrow$ `ks`$(1/2, k_0)$;
**return** $x_0 + dt * k_1$;

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

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

$$
\begin{aligned}
x_{i+1} &\approx x_i + \frac{\Delta t}{2} * \overbrace{f(x_i, t_i)}^{\dot{x}(t_i)} + \frac{\Delta t}{2} * \overbrace{f(x_i + (\Delta t) f(x_i, t_i), t + \Delta t)}^{\approx \dot{x}(t+\Delta t)} \\
&\approx x_i + \frac{\Delta t}{2} k_0 + \frac{\Delta t}{2} k_1
\end{aligned}
$$

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

continued

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

**function** $ks(c, k)$ **is**
    $x \leftarrow \texttt{euler-step}(k, c * dt, x_0)$;
    **return** $f(x,\ t + c * dt)$;
$k_0 \leftarrow f(x_0, t_0)$;
$k_1 \leftarrow \texttt{ks}(1, k_0)$;
**return** $x_0 + dt/2 * (k_0 + k_1)$;

# Exercise: Runge-Kutta 4

$$x_{i+1} \approx x_i + \frac{\Delta t}{6}\overbrace{k_0}^{\dot{x}(t_i)} + \frac{\Delta t}{3}\overbrace{k_1}^{\approx \dot{x}(t_i + \frac{\Delta t}{2})} + \frac{\Delta t}{3}\overbrace{k_2}^{\approx \dot{x}(t_i + \frac{\Delta t}{2})} + \frac{\Delta t}{6}\overbrace{k_3}^{\approx \dot{x}(t_i + \Delta t)}$$

where:

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

# Exercise: Runge-Kutta 4

continued

# Euler Integration

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

**if** $t_0 \geq t_n$ **then** `// Base Case`
  **return** $x_0$;
**else** `// Recursive Case`
  $dx \leftarrow f(x_0, t_0)$;
  $x \leftarrow$ `euler-step`$(dx, dt, x_0)$;
  **return** `int-euler`$(f, t_0 + dt, t_n, dt, x)$;

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

# RK-2 Integration

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

```
if t0 ≥ tn then // Base Case
    return x0;
else // Recursive Case
    x ← rk2-heun(f, t0, x0, dt);
    return int-rk2(f, t0 + dt, tn, dt, x);
```

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

# Multi-method RK Integration

---

**Procedure** integrate$(s, f, t_0, t_n, dt, x_0)$

---

# 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.

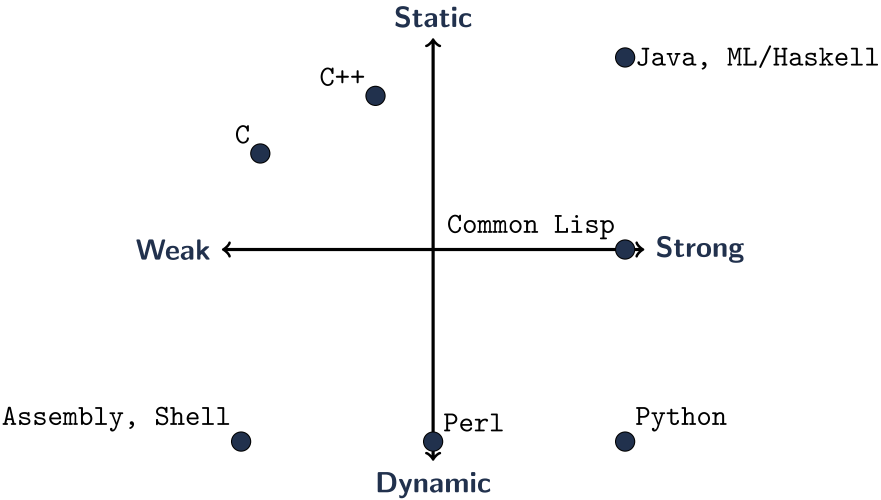
## Examples

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

# Data Type Systems

- Type Checking
  - 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")

# Comparison of Language Type Systems



# Machine Words – Representing Data

`word`

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |

31 0

`unsigned` 42 $\rightsquigarrow$ `0x2a`

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

31 0

`signed` -42 $\rightsquigarrow$ `0xffffffd6`

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

31 0

`float` `42.` $= 1.3125 * 2^5 \rightsquigarrow$ `0x42280000`

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

31 0

# Words and Types

`word`

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |

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

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

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

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

# Type Tags

$$\overset{31}{} \overbrace{1\;1\;0\;0\;0\;0\;0\;0\;0\;1\;0\;0\;1\;0\;0\;1\;0\;0\;0\;0\;1\;1\;1\;1\;1\;1\;0\;1\;0}^{\text{Data}}\;\overbrace{0\;0\;0}^{\text{Tag}}$$

31 … 0

## SBCL Tags (32-bit)

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

$$\overbrace{\texttt{0x180921FA}}^{\text{data}} \times \overbrace{\texttt{000b}}^{\text{tag}} \quad \overset{\text{even fixnum}}{\rightsquigarrow} \quad (\texttt{0x180921FA} >> 2)$$

$$\rightsquigarrow \quad 806503412$$

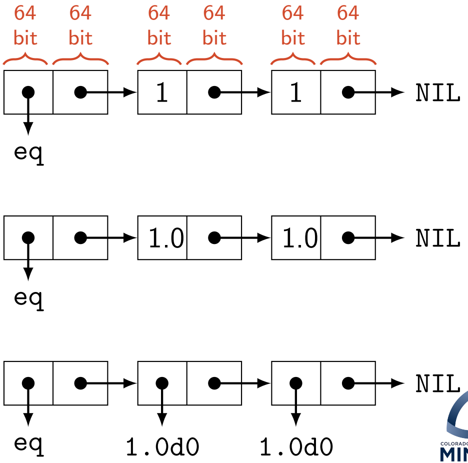
# Example: Tagged Storage

64-bit SBCL:

Fixnum: `(eq 1 1)` $\rightsquigarrow$ `t`

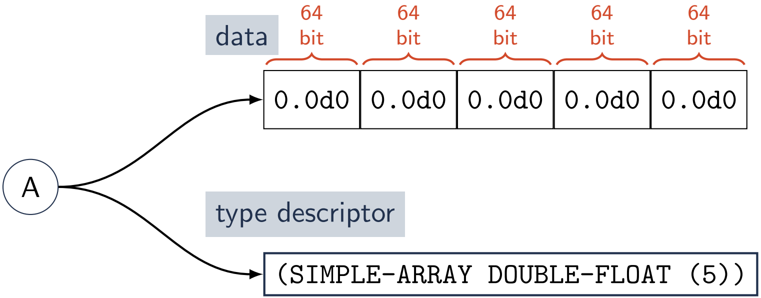
Single Float: `(eq 1.0s0 1.0s0)` $\rightsquigarrow$ `t`

Double Float: `(eq 1.0d0 1.0d0)` $\rightsquigarrow$ `nil`



# Example: SBCL Arrays

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



# Manual Memory Management

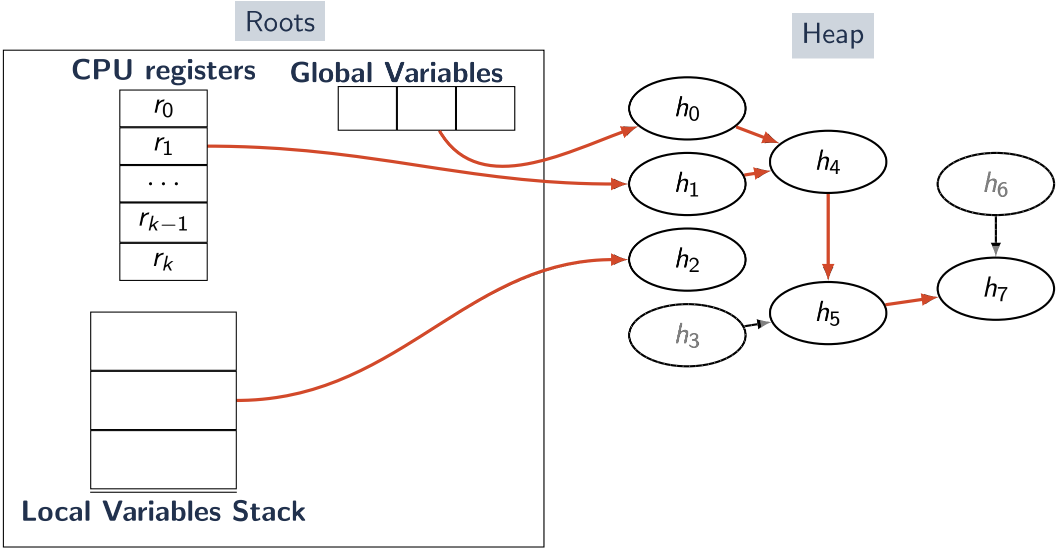
## malloc(n)

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

# Functional Programming Features

- Functions are first class object
- Prefer immutable state
- Garbage collection

# 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->a + 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);
```

# Closure in Lisp

**Local Function**

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

**Lambda Expression**

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

# Example: Recursion

## Iterative

**Function** accumulate(S)

$a \leftarrow 0$;
$i \leftarrow 0$;
**while** $i < |S|$ **do**
  $a \leftarrow a + S_i$;
**return** $a$;

## Recursive

**Function** accumulate(S)

**if** $S$ **then** // Recursive Case
  **return car**$(S) + \mathrm{accumulate}\,(\mathbf{cdr}\,(S))$;
**else** // Base Case
  **return** 0;

# Example: Recursive Accumulate in Lisp

**Recursive Implementation of Accumulate**

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

# Example: Recursive Accumulate Execution Trace

**Recursive Implementation of Accumulate**

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

```
(accumulate '(1 2 3))
        ⇝
(+ 1 (accumulate '(2 3)))
        ⇝
(+ 1 (+ 2 (accumulate '(3))))
        ⇝
(+ 1 (+ 2 (+ 3 (accumulate nil))))
        ⇝
(+ 1 (+ 2 (+ 3 0)))
```

# Exercise: Recursive 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)

# Map

## Definition (map)

Apply a function to every member of a sequence.

$$\mathrm{map} : \underbrace{(\mathbb{D} \mapsto \mathbb{R})}_{\text{function}} \times \underbrace{\mathbb{D}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{R}^n}_{\text{result}}$$

| Function Application |
|---|
| $(f(s_1), f(s_2), \ldots, f(s_n))$ |

# Map Pseudocode

## Procedural

**Function** map(f,S)

**foreach** $s_i \in S$ **do**
  $r_i \leftarrow f(s_i)$;
**return** $r$;

## Recursive

**Function** map(f,S)

**if** $S$ **then** // Recursive Case
  $a \leftarrow f\,(\mathbf{car}(S))$;
  $b \leftarrow \mathrm{map}\,(f,\ \mathbf{cdr}\,(S))$;
  **return** **cons**$(a, b)$
**else** // Base Case
  **return** $()$;

# Map in Lisp

## Map in Lisp

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

# Example: A Map Implementation

### Example Implementation of Map

```lisp
(defun mymap (function list)
  (labels ((helper (list)
             (if list
                 ;; Recursive Case:
                 (cons (funcall function (car list))
                       (helper (cdr list)))
                 ;; Base Case:
                 nil)))
    (helper list)))
```

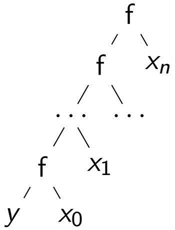
# Fold-left

## Definition (fold-left)

Apply a binary function to every member of a sequence and the result of the previous call, starting from the left-most (initial) element.

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

### Function Application



# Fold-left Pseudocode

## Procedural

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

$i \leftarrow 0$;
**while** $i < |X|$ **do**
  $y \leftarrow f(y, X_i)$ ;
**return** $y$;

## Recursive

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

**if** $X$ **then** // Recursive Case
  $y' \leftarrow f\,(y, \mathbf{car}(X))$;
  **return** fold-left $(f, y', \mathbf{cdr}\,(X))$;
**else** // Base Case
  **return** $y$;

# Fold-left in Lisp

**Fold-Left in Lisp**

```lisp
(reduce #'+                ; function
        '(1 2 3)           ; sequence
        :initial-value 0)
;;; Result 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)

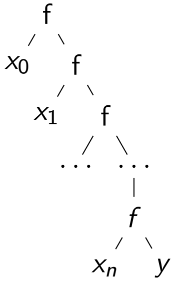
# Fold-right

## Definition (fold-right)

Apply a binary function to every member of a sequence and the result of the previous call, starting from the right-most (final) element.

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

### Function Application



# Fold-right Pseudocode

## Procedural

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

```
i ← |X| − 1;
while i ≥ 0 do
    y ← f(X_i, y) ;
return y;
```

## Recursive

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

```
if X then // Recursive Case
    y′ ← fold-right (f, y, cdr (X));
    return f (car(X), y′);
else // Base Case
    return y;
```

# Fold-right in Lisp

**Fold-Right in Lisp**

```
(reduce #'-                  ; function
        '(2 3)               ; sequence
        :initial-value 1
        :from-end t)
;;; Result 0
```

# MapReduce

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

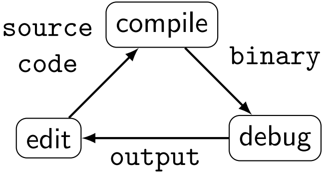
**Function** MapReduce(f,g,X)

$Y \leftarrow$ parallel-map$(f, X)$;
**return** reduce$(g, Y)$;

# Outline

# Lisp Programming Environment

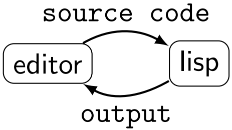
## C Programming




## Lisp Programming

edit,compile,debug

## Lisp on unix



# 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 |

- See `SLIME` drop-down in menu bar for more
- `https://common-lisp.net/project/slime/doc/html/`

# Summary

# Outline

# L'Hôpital's Rule

Evaluate limits using derivatives when $\frac{f(a)}{g(a)} \rightsquigarrow \frac{0}{0}$ (similarly for $\infty$):

$$\left(\left(\lim_{x \to a} f(x) = 0\right) \wedge \left(\lim_{x \to a} g(x) = 0\right)\right) \implies \left(\lim_{x \to a} \frac{f(x)}{g(x)} = \lim_{x \to a} \frac{f'(x)}{g'(x)}\right)$$

# LET

- Creates a new scope and variable bindings

# Example: LET

## C Block Scope

**C**

```c
{
    int a = 1;
    int b = 1;
    printf("(%d %d)\n",
           a, b);
}
```

**Lisp**

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

**Output**

```
(1 2)
```

# Example: LET

## Scope Nesting

**C**

```c
{
    int a = 1;
    printf("%d\n", a);
    {
        int a = 2;
        printf("%d\n", a);
    }
    printf("%d\n", a);
}
```

**Lisp**

```lisp
(let ((a 1))
  (print a)
  (let ((a 2))
    (print a))
  (print a))
```

**Output**

```
1
2
1
```

# Example: LET

"Parallel" assignments

## Lisp

```
(let ((a 1)
      (b 2))
  (let ((a 3)
        (b a))
    (print (list a b))))
```

## Output

```
(3 1)
```

# Example: LET*

"Consecutive" assignments

**Lisp**

```
(let ((a 1)
      (b 2))
  (let* ((a 3)
         (b a))
    (print (list a b))))
```

**Output**

```
(3 3)
```

# DOTIMES

- Iterate a for $n$ steps

# Example: DOTIMES

**C**

```c
for( int i = 0; i < 5; i ++ ) {
      printf("%d",i);
}
```

**Lisp**

```lisp
(dotimes (i 5)
  (print i))
```

**Output**

```
0
1
2
3
4
```

# DOLIST

- Iterate over a list

# Example: DOLIST

**Lisp**

```
(dolist (x '(a b c))
  (print x))
```

**Output**

```
A
B
C
```

# Example: LOOP

counting

**Lisp**

```
(loop for i below 5
      do (print i))
```

**Output**

```
0
1
2
3
4
```

# Example: LOOP

list iteration

**Lisp**

```lisp
(loop for x in '(a b c)
      do (print x))
```

**Output**

```
A
B
C
```

# Example: LOOP

collecting

**Lisp**

```
(let ((x (loop for i below 5
               when (evenp i)
               collect i)))
  (print x))
```

**Output**

```
(0 2 4)
```

# Example: REDUCE

collecting

**Lisp**

```lisp
(let ((x (reduce (lambda (a x)
                   (if (evenp x)
                       (cons x a)
                       a))
                 '(4 3 2 1 0)
                 :initial-value nil)))
  (print x))
```

**Output**

```
(0 2 4)
```

# Case

- Control structure
- Selects clause that matches the test argument

# Example: CASE

**C**

```c
switch('B') {
case 'A':
    puts("Got A");
    break;
case 'B':
    puts("Got B");
    break;
case 'C':
    puts("Got C");
    break;
}
```

**Lisp**

```lisp
(case 'b
  (a (print "Got A"))
  (b (print "Got B"))
  (c (print "Got C")))
```

**Output**

```
Got B
```

# Example: S-Expression to XML

## Lisp

```lisp
(labels ((visit (e i)
           (if (listp e)
               (progn
                 ;; opening tag
                 (format t "~&<~A>" (car e) )
                 ;; Recurse on arguments
                 (dolist (e (cdr e))
                   (visit e (+ i 2)))
                 ;; Closing tag
                 (format t "~&</~A>" (car e)))
             ;; Else, print the element
             (format t "~&~A" e))))
  (visit '(and x (or y z)) 0))
```

## Output

```
<AND>
X
<OR>
Y
Z
</OR>
</AND>
```

# Example: S-Expression to XML w/ indentation

## Lisp

```lisp
(labels ((visit (e i)
           (let ((indent (make-string i
                                      :initial-element #\Space)))
             (if (listp e)
                 (progn
                   ;; opening tag
                   (format t "~&~A<~A>" indent (car e) )
                   ;; Recurse on arguments
                   (dolist (e (cdr e))
                     (visit e (+ i 2)))
                   ;; Closing tag
                   (format t "~&~A</~A>" indent (car e)))
                 ;; Else, print the element
                 (format t "~&~A~A" indent e)))))
  (visit '(and x (or y z)) 0))
```

## Output

```
<AND>
  X
  <OR>
    Y
    Z
  </OR>
</AND>
```