Lisp Introduction

Dr. Neil T. Dantam

CSCI-534, Colorado School of Mines

Spring 2020

What is Lisp?

Definition (Lisp)

A family of programming languages based on s-expressions.

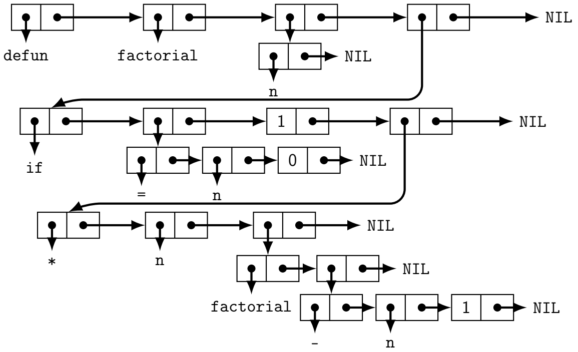
"Math"

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

Code

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

Data Structure



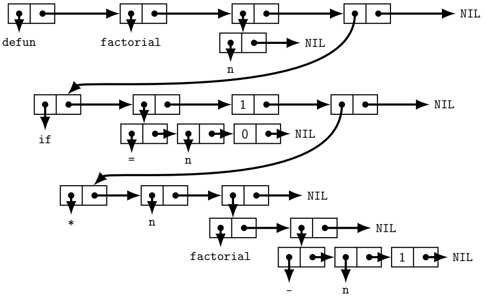
Homoiconic

Code is Data

Code

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

Data Structure



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

Introduction

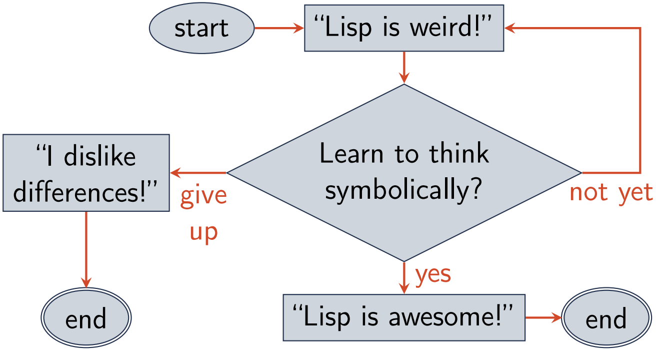
Why study lisp?

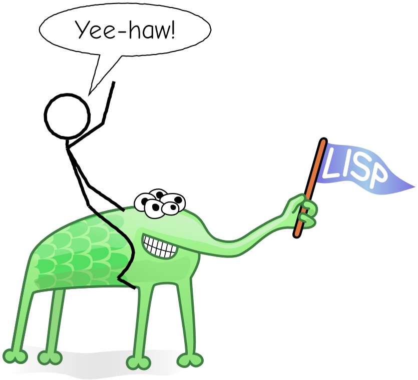
- Functional Programming:
  - Planning algorithms often are functional/recursive
  - Lisp has good support for functional programming.
- Symbolic Computing:
  - Planning algorithms must often process symbolic expressions
  - Lisp has good support for symbolic processing
- Understanding the abstractions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



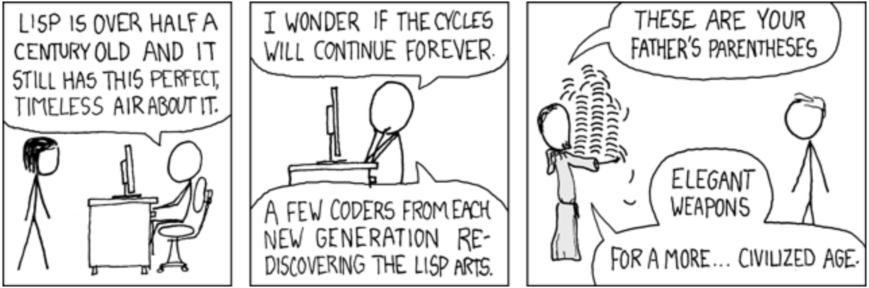


Lisp might feel like it's "from outer space" (at first).

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` — 1: integer, 1.0: float
- (eq 1 1.0) ⇝ `nil` — 1: integer, 1.0: float
- (eql 1 1.0) ⇝ `nil` — 1: integer, 1.0: float
- (equal 1 1.0) ⇝ `nil` — 1: integer, 1.0: float
- (equalp 1 1.0) ⇝ `t` — 1: integer, 1.0: float

- `(= "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) $\rightsquigarrow$
- (not 1) $\rightsquigarrow$
- (eq t (not nil)) $\rightsquigarrow$
- (eq t 1) $\rightsquigarrow$
- (eq nil (not 1)) $\rightsquigarrow$
- (eq nil (not "a")) $\rightsquigarrow$

- (eq (list "a" "b") (list "a" "b")) $\rightsquigarrow$
- (equal (list "a" "b") (list "a" "b")) $\rightsquigarrow$

- (eq (list "a" "b") (list "a" "B")) $\rightsquigarrow$
- (equal (list "a" "b") (list "a" "B")) $\rightsquigarrow$

- (equalp (list "a" "b") (list "a" "B")) $\rightsquigarrow$

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)

return $n + 1$;

Common Lisp

function name function arguments

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

result

Exercise: Function Definition
Sine Cardinal

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

Pseudocode

Procedure sinc(θ)

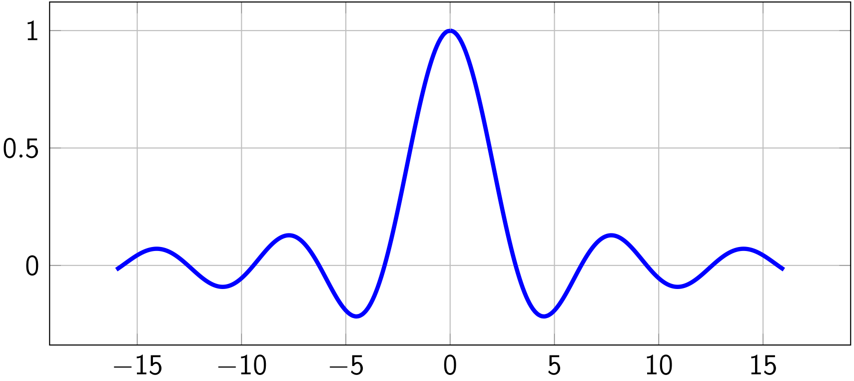
return $\sin(\theta)/\theta$;

Common Lisp

What's wrong (mathematically) with this definition?

Limit of sinc θ

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



Conditional

IF

IF

Conditional execution based on a single test:

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

Pseudocode

Procedure even?(n)

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

Common Lisp

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

test: `(= 0 (mod n 2))`

then clause: `t`

else clause: `NIL`

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(θ)

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

Common Lisp

Taylor Series

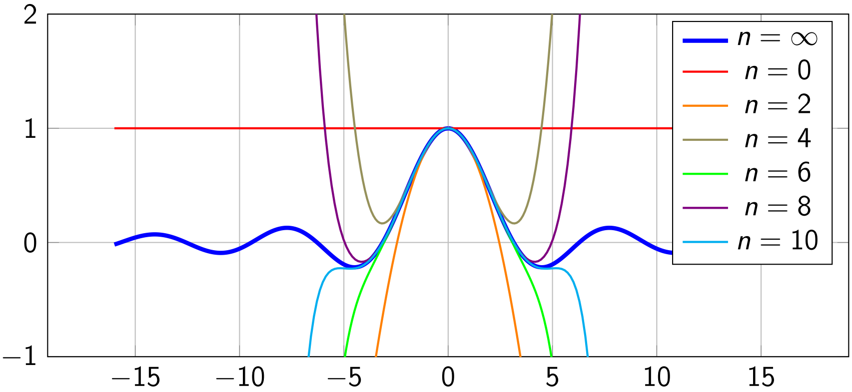
Represent function $f(x)$ as infinite sum of derivatives around point a:

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

Polynomial approximation of functions

Sinc Taylor Series

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



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)

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

Common Lisp

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

Exercise: Conditionals

COND

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

Pseudocode

Procedure sinc(θ)

```
if 0 = θ then
    return 1;
else if θ^2 < .00001 then
    return
      1 − θ^2/6 + θ^4/120;
else
    return sin(θ)/θ;
```

Common Lisp

Recursion

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

Base Case: Terminating condition

Recursive Case: Reduction towards the base case

Example: Factorial

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

Procedure factorial(n)

```
if 0 = n then // base case
    return 1;
else // recursive case
    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)

```
if 0 = n then // base case
    return 1;
else // recursive case
    return
      n * factorial(n - 1);
```

Common Lisp

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

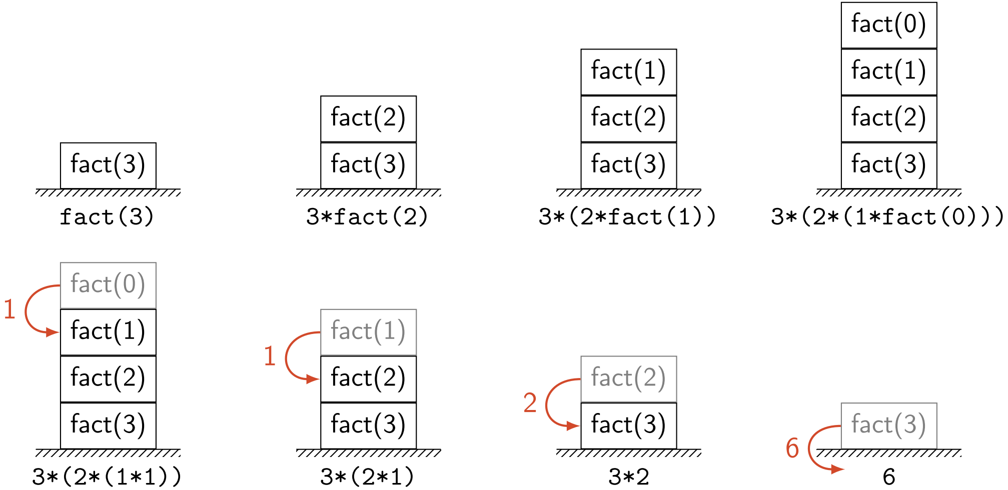
Factorial Execution Trace

Procedure factorial(n)

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

```
factorial(3)
3*factorial(2)
3*(2*factorial(1))
3*(2*(1*factorial(0)))
3*(2*(1*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	Common Lisp

Exercise: Recursion, Accumulate

Iterative

Function accumulate(S)

```
a ← 0;
i ← 0;
while i < |S| do
    a ← a + S_i;
return a;
```

Recursive

Exercise: Recursion, Accumulate

Lisp Code

Exercise: Recursion, Accumulate

Execution Trace

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->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);
```

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

```lisp
(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

```lisp
(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

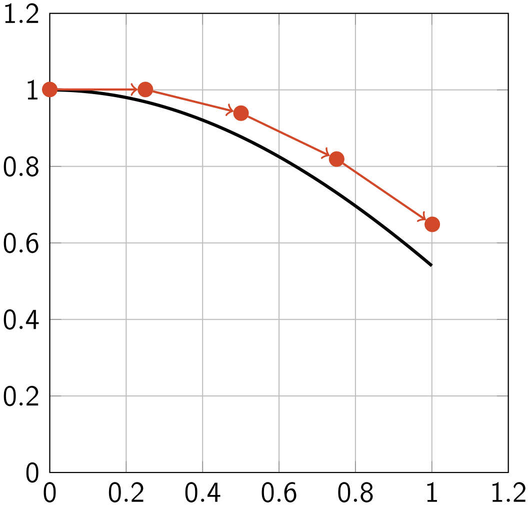
Given:
- Derivative: $\frac{d}{dt}x(t) = f(x, t)$
- Initial time: t_0
- Final time: t_n
- Initial value: $x(t_0)$

Find: $x(t_n)$

Solution: Follow derivative along discrete time intervals Δt 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)

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

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

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

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

Averaging derivatives at current and next point.

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

continued

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

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

Exercise: Runge-Kutta 4

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

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

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

Euler (RK-1) Integration

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

if $t_0 \geq t_n$ **then** `// Base Case`
  return x_0;
else `// Recursive Case`
  let
    $dx \leftarrow f(x_0, t_0)$;
    $x \leftarrow$ `euler-step`(dx, dt, x_0);
    $t_1 \leftarrow t_0 + dt$;
  in return `int-rk1`(f, t_1, t_n, dt, x);

```lisp
(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)

if $t_0 \geq t_n$ **then** // Base Case
  return x_0;
else // Recursive Case
  let
    $x \leftarrow$ rk2-heun(f, t_0, x_0, dt);
    $t_1 \leftarrow t_0 + dt$;
  in return int-rk2(f, t_1, t_n, dt, x);

```lisp
(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

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)

return $g(42, a)$;

Example (Returning)

Function f(a)

function $g(b)$ **is**
  return $a + b$;
return g;

Counterexample

Function f(a,b)

return $a + b$;

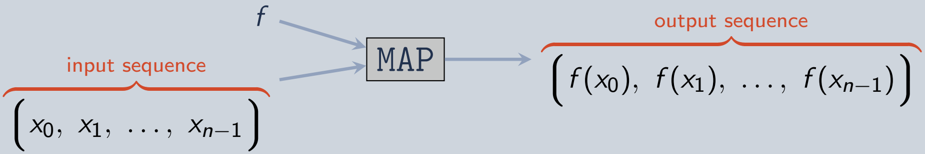
Map function

Definition (map)

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

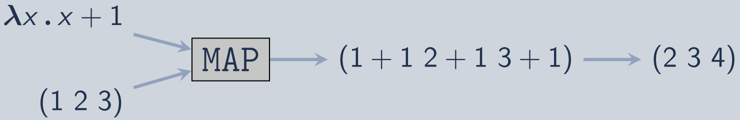
$$\texttt{map} \;:\; \underbrace{(\mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{X}^n}_{\text{input sequence}} \mapsto \underbrace{\mathbb{Y}^n}_{\text{output sequence}}$$

Illustration

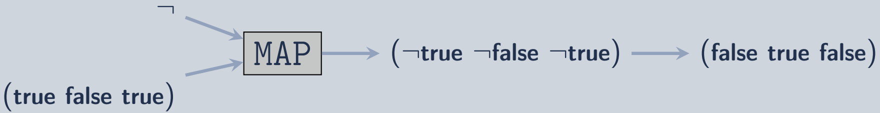


Example: Map

+1



¬



Algorithm: Map function

Functional Map

Procedure map(f,s)

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

Imperative Map

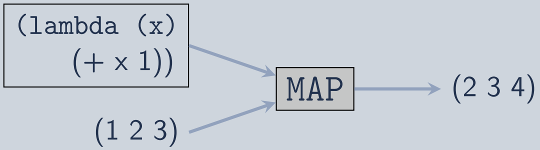
Procedure map(f,s)

```
n ← length(s);
Y ← make-sequence(n);

i ← 0;
while i < n do
    Y[i] ← f(s[i]);
    i ← i + 1;
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)
```

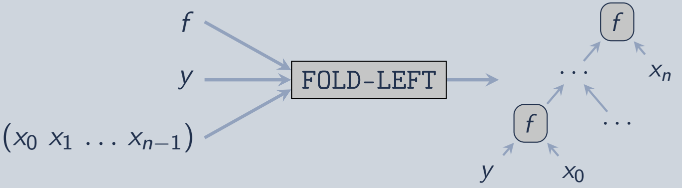
Fold-left

Definition (fold-left)

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

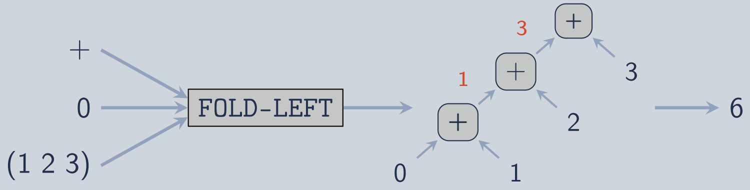
$$\text{fold-left} : \underbrace{(\mathbb{Y} \times \mathbb{X} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init.}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

Illustration

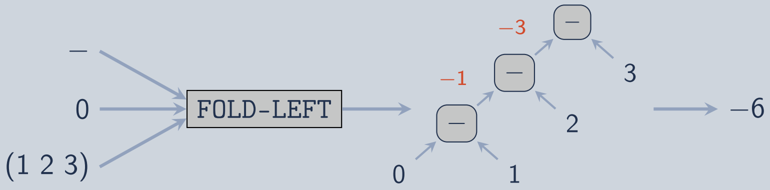


Example: Fold-left

Example (Addition)



Example (Subtraction)



Algorithm: Fold-left

Imperative

Function fold-left(f, y, X)

$i \leftarrow 0$;

while $i < |X|$ **do**

  $y \leftarrow f(y, X_i)$;

  $i \leftarrow i + 1$;

return y;

Functional

Function fold-left(f, y, X)

if `empty`(X) **then** **return** y;

else

  let $y' \leftarrow f\left(y, \texttt{first}(X)\right)$ **in**

    return `fold-left` $(f,\ y',\ \texttt{rest}(X))$;

Example: Fold-left in Lisp

Example (Addition)

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

Example (Subtraction)

```lisp
(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

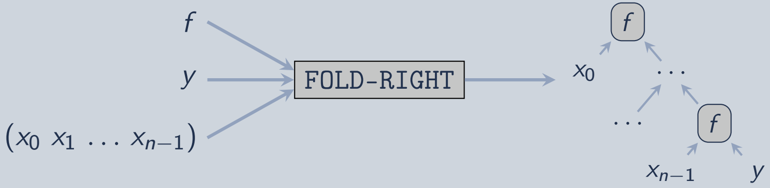
Fold-right

Definition (fold-right)

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

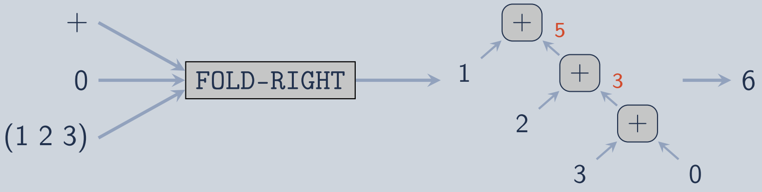
$$\text{fold-right} : \underbrace{(\mathbb{X} \times \mathbb{Y} \mapsto \mathbb{Y})}_{\text{function}} \times \underbrace{\mathbb{Y}}_{\text{init.}} \times \underbrace{\mathbb{X}^n}_{\text{sequence}} \mapsto \underbrace{\mathbb{Y}}_{\text{result}}$$

Illustration

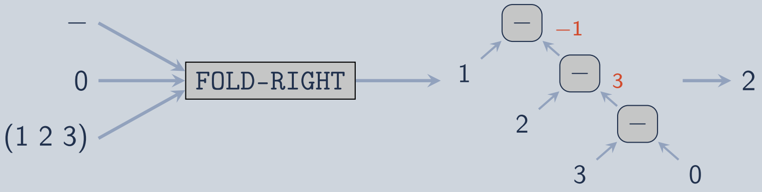


Example: Fold-right

Example (Addition)



Example (Subtraction)



Algorithm: Fold-right

Procedural

Function fold-right(f,y,X)

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

Recursive

Function fold-right(f,y,X)

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

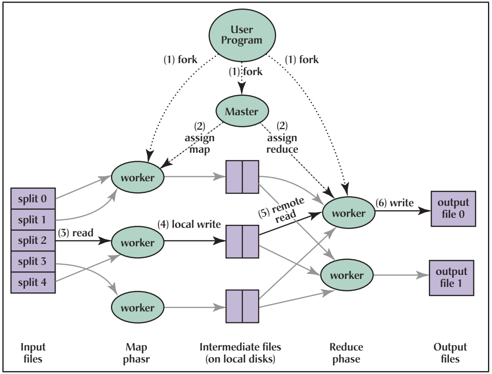
```
(reduce #'-
        '(1 2 3)
        :initial-value 0
        :from-end t)
;;; => (- 1 (- 2 (- 3 0)))
;;; => 2
```

Application: MapReduce

Function MapReduce(f,g,X)

$Y \leftarrow$ parallel-map(f, X);
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.

MINES

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:
  - `int` $\times$ `string`
  - `float`^{4}
- Function:
  - `int` $\times$ `int` $\mapsto$ `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

`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 ⇝ `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 ⇝ `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.` $= 1.3125 * 2^5$ ⇝ `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

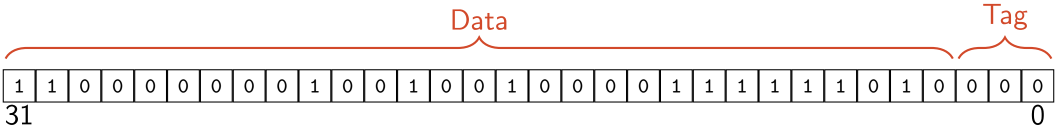
$\texttt{0xc0490fd0} \overset{?}{\leadsto} -1068953648$ (signed)

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

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

$\texttt{0xc0490fd0} \overset{?}{\leadsto}$ `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

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

$$\rightsquigarrow 806503412$$

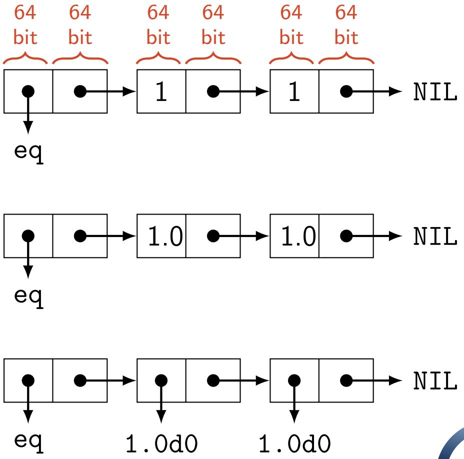
Example: Tagged Storage

64-bit SBCL:

Fixnum: `(eq 1 1)` ⇝ `t`

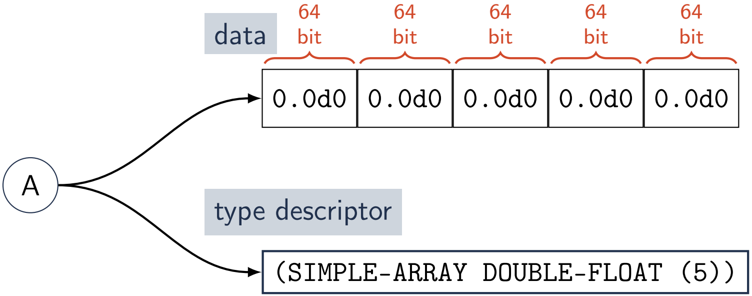
Single Float: `(eq 1.0s0 1.0s0)` ⇝ `t`

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



Example: SBCL Arrays

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



Manual Memory Management

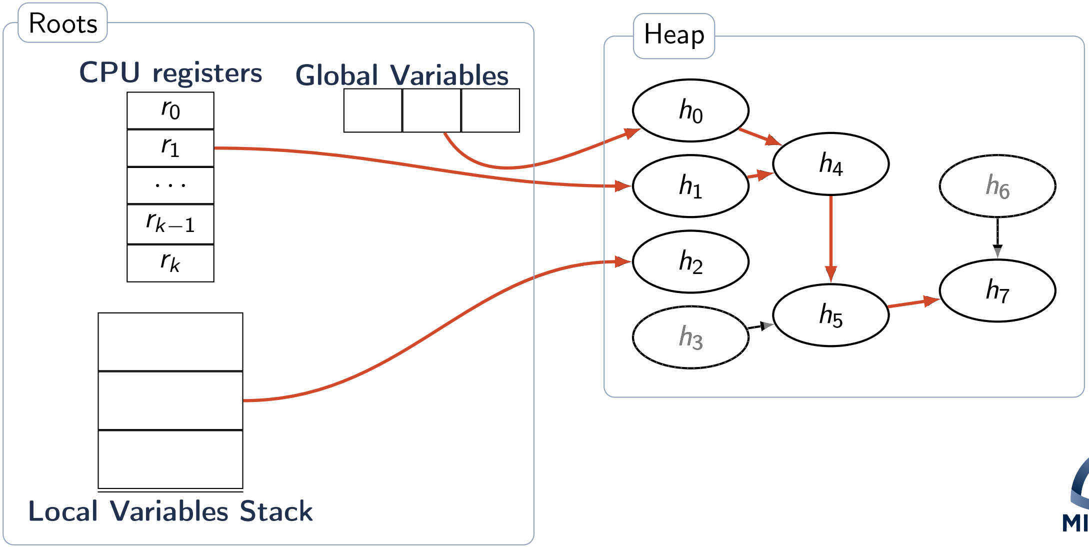
`malloc(n)`

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

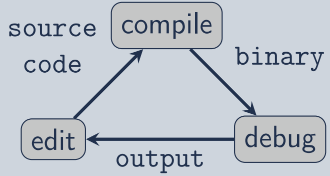


Outline

Lisp Programming Environment

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

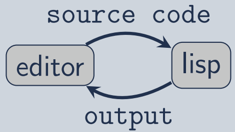
C Programming



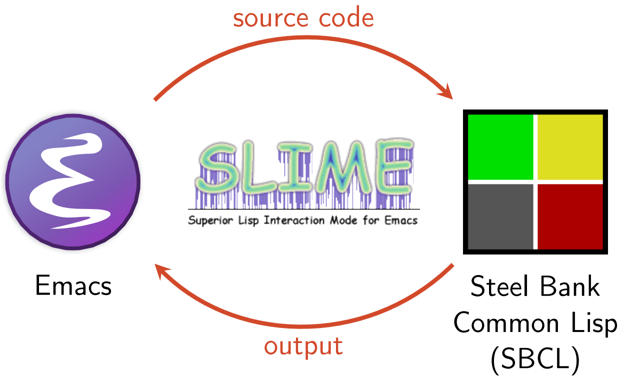
Lisp Programming

edit,compile,debug

Lisp on unix



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

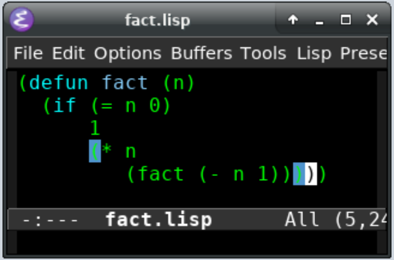
```
(defun foo (x)
  (+ 1 x))
```

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*.