Racket

Zena M. Ariola
Some parts taken from Dan Grossman’s lecture notes

May 30, 2014
Lisp (List Processor) was developed in the late 50s for research in AI and symbolic computation.

List was very innovative if you think that Frontan at the same time did not even have recursion. LISP not only had recursion, he also had GC. It took many years to have that idea in industrial languages (e.g. Java)

Abstract view of memory: cons cells instead of array of numbered locations

Really minimalist syntax
- Programs and data have the same representation

- Like ML, functional focus with imperative features
  - Anonymous functions, closures, higher-order functions
  - Does not have pattern matching

- Unlike ML, no static type system: accepts more programs, but most errors do not occur until run-time

- The basic constructs of the language are expressions, namely, syntactic entities that return a value. Statements are entities that are executed to change the machine state (Fortran was a statement oriented language)
Syntax

- \((+ 4 5)\)
- \((+ 4 5 1)\)
- \((+ (* 2 3)(* 4 5))\)
- \((\text{function arg1 arg2 \ldots argn})\)

- Special forms:
  - \((\text{cond (p1 e1) (p2 e2) \ldots (pn en)})\)
  - \((\text{cond (\(\leq 1 2\) 6) (\(= 1 2\) 7)})\)
  - \((\text{cond (\(\leq 2 1\) 2) (\(\leq 3 2\) 3)})\); returns undefined
  - \((\text{cond (diverge 1) (#t 0)})\); diverge
  - \((\text{cond (#t 0) (diverge 1)})\); returns 0
Data Structure

- pair (address and decrement) – cons cell
- `(cons 1 2) '(1 . 2)
- `(cons (+ 1 2) 6) '(3 . 6)
- `(cons 1 (cons 2 (cons 3 null)) ) ) '(1 2 3)
- `(list 1 2 3)
- `(cons 1 a) ; Error
- `(cons 1 'a)
- `(list 1 #t 'banana "foo")
(define (fact n)(if (= n 0) 1 (* n (fact (- n 1))))))
(define (fact n)(if (= n 0) (1)(* n (fact (- n 1))))))
(define (fact n)(if = n 0 1 (* n (fact (- n 1))))))
(define fact (n)(if (= n 0) 1 (* n (fact (- n 1))))))
(define (fact n)(if (= n 0) 1 (n * (fact (- n 1))))))
Scope

- `(let ( (x 5) (y 9) ) (+ x y) )`
- `(let ( (x 5) (y (+ x x))) 3 )`
- `(let* ( (x 5) (y (+ x x))) (+ x y) )`
- `(letrec ( (f (lambda (z)(...g...)))
             (g (lambda (w)(...f...))))
             (f 99))`
- `(letrec ( (x y) (y x)) x ) – undefined`
- `(letrec ( (x y) (y x)) (+ x 1) ) – error`
Functions are first class citizens

- `(lambda (x) (+ x 1)) ;; anonymous function`
- `(define inc (lambda (x) (+ x 1)))`
  `inc 2`
- `(define a (cons (lambda (x) x) inc))`
  `((cdr a) 1)`
Functions (cont.)

- Function in Racket takes 0, 1 or more arguments

- No Currying: `(define (f x y)(+ x y))`
  
  `(f 2 3)`
  `(f 2) ; error`
  `((f 2) 3) ; error`

- `(define f (lambda () 3))`
  
  `(f)`
  `((f)) ; error`

- `(define apply (lambda (f)(lambda (x)( f x ))))`
  
  `((apply (lambda (x)(+ x 1))) 2)`
  `(apply (lambda (x)(+ x 1)) 2) ; error`
Functions are first class citizens

- (define map (lambda (f x)
    (cond
      ((null? x) null)
      (else (cons (f (car x))(map f (cdr x))))))
  ))

- (map (lambda (x)(+ x x)) '(1 2 3 4))
Static vs Dynamic scope

- (define b 3)
- (define f (lambda (x) (* 1 (+ x b))))
- (let ((b 99)) (f 0))
Dynamic typing

datatype int_or_string = I of int | S of string
fun sum xs = case xs of
  [] => 0
  | (I i) :: xs' => i + sum xs'
  | (S s) :: xs' => String.size s + sum xs'

(define (sum xs)
  (cons [(null? xs) 0]
    [(number? (car xs))(+ (car xs)(sum (cdr xs)))]
    [(string? (car xs))(+ (string-length (car xs))
                              (sum (cdr xs)))]
    [#t (error "expected an int or a string")]))
Dynamic typing

datatype exp = Const of int | Add of exp * exp
fun eval e = case e of
  Const n => n
| Add (e1,e2) => (eval e1) + (eval e2)
(define (Const n)(list 'Cons n))
(define (Add e1 e2)(list 'Add e1 e2))
(define (Const? x)(eq? (car x) 'Const))
(define (Add? x)(eq? (car x) 'Add))
(define (Const-int e)(car (cdr e)))
(define (Add-e1 e)(car (cdr e)))
(define (Add-e2 e)(car (cdr (cdr e))))
(define (eval e)
  (cond [(Const? e) (Const-int e)]
     [(Add? e)(let ([v1 (eval (Add-e1 e))]
                   [v2 (eval (Add-e2 e))])
                 (Const (+ v1 v2)))]
     [#t (error "expected an expression")]))

(eval (Add (Const 1)(Const 3)))
Racket is ML with one datatype!

```plaintext
datatype Type = Int of int
  | String of string
  | Pair of Type * Type
  | Fun of Type -> Type
  | ....

fun car v = case v of
    Pair (a,b) => a
  | _ => raise error

fun pair? v = case v of
    Pair (a,b) => true
  | _ => false
```
Sharing

\[ ((\lambda (x)(\text{cons } x x )) (\text{cons } 1 2)) \]
Assignment and its impact. Do we need to care if we have multiple pointers to the same cons cell?
Mutable vs immutable data structure.
Programs as data: backquote
Lazy evaluations; Thunks; Delay and force
Memoization
Macros
Assignment

- `(set! x e)` it updates x; assignment in Java, C
- `(define b 3)`
  `(define f (lambda (x) (+ x b)))`
(define b 3)
(define f (lambda (x) (+ b 4)))
(define c (+ b 4)) ;; 7
(set! b 5);
(define z (f 4)) ; 9
(define w c) ; 7
(define b 3)
(define f (lambda (x) (+ x b)))

Copy before it changes!
(define f (let ([bb b])
            (lambda (x) (+ x bb)))))
(define increment-and-return1
  (let ([v 0])
    (lambda (x) (set! v (+ x v)) v)))

(define increment-and-return2
  (lambda (x)
    (let ([v 0]) (set! v (+ x v)) v)))
Variables are not values

(define a 99)
(define f (lambda (x) (set! x 0)))
(f a)
a
Box and Unbox

(define a (box 99))
(define f (lambda (x) (set-box! x 0)))
(f a)
(unbox a)
Mutable cons

(define a (mcons 1 2))
(define b ((lambda (x)(cons x x )) a ))
(define c (car b))
(set-mcar! c 99)
(define a (mcons 1 2))
(define f (lambda (x) (set-mcar! x 99)))
(f a)
Lazy evaluation - Thunks

- You want to avoid expensive computations. You turn an expression \( e \) into \((\text{lambda } () e)\).

- \((\text{lambda } (\text{th}) (+ (\text{th}) (\text{th}) (\text{th})))\)

- Compute only if needed and only once.
Delay and force

- `(define (my-delay th) (mcons #f th))`
- `(define (my-force p) (if (mcar p) (mcdr p)
   (begin (set-mcar! p #t)
   (set-mcdr! p ((mcdr p))
   (mcdr p)))))`
Promises

- (define (f th)(+ (th) (th)))
  (f (lambda ()(display "hi")(+ 2 3)))

- (define (g p)(+ (my-force p) (my-force p)))
  (g (my-delay (lambda ()(display "hi")(+ 2 3))))
- Use the concept of a thunk to create streams (infinite lists).
- Use mutation to create memoization: avoid recomputing a value of a function.
Macros

- Expand `(my-if e1 then e2 else e3)` to `(if e1 e2 e3)`
- Expand `(comment-out e1 e2)` to `e2`
- Expand `(my-delay e)` to `(mcons #f (lambda () e))`
Expand head to car
  - Rewrite (+ headt foo) to (+ cart foo)
  - Rewrite head-door to car-door

Expand ADD(x,y) to x + y
  - ADD(1,2/3)*4 means 1 + 2/3*4
  - ADD(1,2/3)*4 means (1 + 2/3)*4
(define (f x)(let ((y 1))(+ y x)))
(let ((y 7))(f y))
(let ((y 7))(let ((y 1))(+ y y)))

(define (f x)(+ y x))
(let ((y 7))(f y))
(let (y 7)(+ y y ))