Racket

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

October 9, 2013
Lisp (List Processor) was developed in the late 50s for research in AI and symbolic computation.

List was very innovative if you think that Fortran at the same time did not even have recursion. LISP not only had recursion, he also has 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

- 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

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

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

- (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))))))

- (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)`
- `(letrec ((x y) (y x)) (+ x 1))`
Top-level

- The bindings in a file work like local defines, i.e., letrec.

- Like ML, you can refer to earlier bindings:
  
  ```
  (define a (+ 2 3))
  (define b (+ a 1))
  ```

- Unlike ML, you can also refer to later bindings. But refer to later bindings only in function bodies:
  
  - Because bindings are evaluated in order.
  - Detail: Will get an error instead of undefined.
  - Unlike ML, cannot define the same variable twice in module. Would make no sense: cannot have both in environment.
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" )
Functions are first class citizens

- (lambda (x)(+ x 1)) ;; anonymous function

- (define inc (lambda (x)(+ x 1)))
  inc 2

- (define twice (lambda (f)(lambda (x)( f ( f x)))))
  ((twice inc) 2)
  (twice inc 2) ; error

- (define a (cons twice inc))
  ((cdr a) 1)

- (define compose (lambda (f g) ( lambda (x)(f ( g x)) ) ) )
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))
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
Static vs Dynamic scope

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

\[(\text{lambda} \ (x) (\text{cons} \ x \ x)) \ (\text{cons} \ 1 \ 2)\]
Higher order functions or functions as first class citizens. You will practice with this concept on the next assignment.

Currying: multi-arguments functions in terms of functions which take one parameter at a time. You will practice with this concepts on the next assignment.

Static (Racket) vs Dynamic scope (Lisp)
Assignments 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 (practice in the new assignment).

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) (* 1 (+ x b)))))
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4)) ;; 7
(set! b 5);
(define z (f 4)) ; 9
(define w c) ; 7
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))

Copy before it changes!
(define f (let ([bb b])
             (lambda (x) (* 1 (+ 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 $(\lambda () e)$.
- $(\lambda (th) (+ (th) (th) (th)))$
- Compute only if needed and only once.
Delay and force

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

► \[(\text{define } (f \ \text{th})(+ (\text{th}) (\text{th}))))\]
\[(f \ (\text{lambda } ()(\text{display } "\text{hi}" )(+ 2 3))))\]

► \[(\text{define } (g \ p)(+ (\text{my-force } p) (\text{my-force } p))))\]
\[(g \ (\text{my-delay} \ (\text{lambda } ()(\text{display } "\text{hi}" )(+ 2 3))))\)]
- Use the concept of a thunk you 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))