More Scheme

CSCE A331

Quiz

1. What is (car ‘((2) 3 4))?
   (2)
2. What is (cdr ‘((2) (3) (4)))?
   ((3)(4))
3. What is (cons ‘2 ‘(2 3 4))?
   (2 2 3 4)
4. What is the length of the list ‘(()()()())?
   4
5. Which element does (car (cdr ‘(x y z))) extract from the list?
   y
6. What do some people thing LISP stands for?
   Losta Insane Stupid Parenthesis

Reading Input

- (read) returns whatever is input at the keyboard

- (define x (read))
  55

x

55

Display Output

- Use display:

- (display (+ 3 4))
- (display x)
- (newline) ; output newline

- Can use to add to functions for debugging or use Trace functions

Trace – Shows function calls

- Choose “Pretty Big” as your language
- Add (require (lib "trace.ss")) to the top of your program
- Add (trace functionname) or (untrace functionname) to turn off tracing

```
> (trace remove-duplicates)
(remove-duplicates)
> (remove-duplicates '(a b c a))
|(remove-duplicates (a b c a))
| (remove-duplicates (b c a))
| |(remove-duplicates (c a))
| | (remove-duplicates (a))
| | |(remove-duplicates ())
| | |()
| | (a)
| |(c a)
| (b c a)
|(b c a)
```

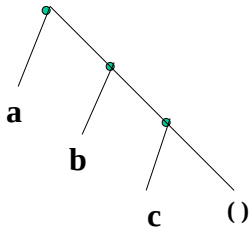
Structure of Lists

- List : a sequence of zero of more elements
- May be heterogeneous
  - (a 20 (20 20) (lambda (x) (+ x x)))
- List of Lists
- (+ 3 4)
  - A list
  - An expression
- Allows expression to be built and then evaluated

Tree structure of lists

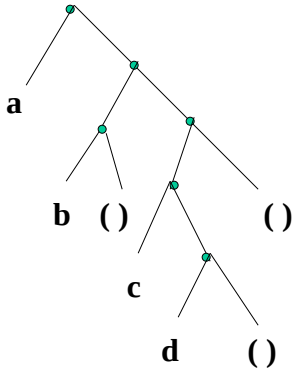
(a b c)

```
(cons a
        (cons b
                (cons c '()))
        )
)
```



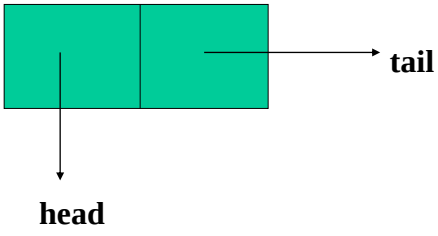
List structure

(a (b) (c d))



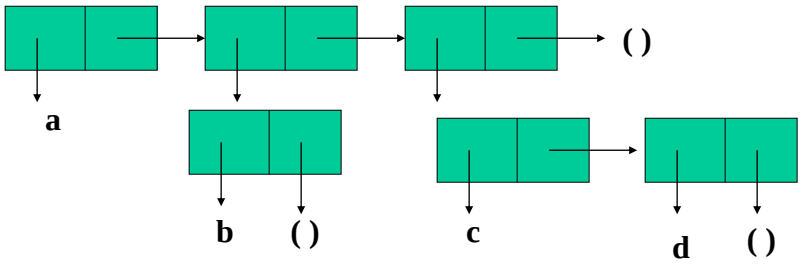
List storage

- Independent of allocation schemes
- Familiarity is helpful for assessment
- *Cons* is the constructor
  - Allocates a single cell



List structure

(a (b) (c d))



Notion of Equality

- Eq?
  - Checks if the two pointers are the same
- Equal?
  - Checks if the two arguments are lists with "equal" elements.
  - Recursive
  - Structurally the same
- Eq? ⇔ equal? for symbols.

Examples

(equal? 'foo 'foo) ⇔ (eq? 'foo 'foo)
⇔ true
(equal? '(a b) '(a b))
⇔ true
(eq? '(a b) '(a b))
⇔ false
(define x '(a b c))
(define y (cons (car x) (cdr x)))
(equal? x y)
⇔ true
(eq? x y)
⇔false

(define mylist '(a b c))
(eq? (member 'b mylist)
(cdr mylist)
)
⇔ true

More List Functions

- **append** takes two arguments and returns the concatenation of two lists. be careful here not to confuse append with cons.

  (append '(a b c) '(d e f)) → (a b c d e f)
  (append '(a b (c)) '((d) e f)) → (a b (c) (d) e f)

- **list** returns a list constructed from its arguments.

  (list 'a) → (a)
  (list 'a 'b 'c 'd 'e 'f) → (a b c d e f)
  (list '(a b c)) → ((a b c))
  (list '(a b c) '(d e f) '(g h i)) → ((a b c)(d e f)(g h i))

More List Functions

- **length** returns the length of a list.

  (length '(a b c d ef gh i jk)) → 8
  (length '(a b c d (ef gh i jk))) → 5

- **reverse** returns the same list, only in reversed order. Note, this is only shallow reverse.

  (reverse '(a b d g o)) → (o g d b a)
  (reverse '(a (b d) g o)) → (o g (b d) a)

Exercise

- Write a helper function to return the first half of a list. Here is the main function:

```
(define (firsthalf lst)
      (getfirsthalf lst (quotient (length lst) 2))
)
```

- Write a helper function to return the second half of a list. Here is the main function:

```
(define (secondhalf lst)
      (getsecondhalf lst (quotient (length lst) 2))
)
```

Merge Method

- Write a method called Merge that merges two sorted lists:

```
(define (Merge x y)



)
```

MergeSort

- Write a MergeSort method that uses your Merge method to sort a list of numbers.

let & let*

```
(let   ((x1 E1)
        (x2 E2)
        (x3 E3)
          ….
        (xn En))
 F)
```

- Expressions E1, E2, … E_n are evaluated
- Evaluate F with x_i's bound to E_i's
- Value of let is the value of F

Local Variables (let & let*)

- Used to factor out common expressions
- Introduce names in subexpressions
- Order of evaluation of expression is undetermined (let)
- Order of evaluation of expression is sequential (let*)

Examples

```
(let ((x 2)
      (y x))
  y)
0
```


```
(let* ((x 2)
       (y x))
  y)
2
```

Tail recursion

A recursive function is **tail-recursive** if

(a) it returns a value without needing recursion
OR

(b) simply the result of a recursive activation

i.e. just return the value at the end

- Can be efficiently implemented
  - Don't need stacks
- Can convert many functions to be tail recursive

Examples

Factorial Function

```
(define factorial
  (lambda(n)
    (cond ((= n 1) 1)
          (else (* n (factorial (- n 1)))))))
```

Examples

Factorial Function: Tail Recursive

```
(define factorial2
  (lambda(n m)
    (cond ((= n 1) m)
          (else (factorial2
                  (- n 1)
                  (* m n))))))
```

```
(define factorial
  (lambda(n)
    (factorial2 n 1)))
```

Examples

Getridof Function (get rid of an item from a list)

```
(define getridof
  (lambda(list item)
    (cond ((null? list) '())
          ((equal? item (car list)) (getridof (cdr list) item))
          (else (cons (car list) (getridof (cdr list) item))))))
```

Examples

getridof Function (Tail recursive)

```
(define gro
  (lambda(list item list2)
    (cond ((null? list) list2)
          ((equal? item (car list)) (gro (cdr list) item list2))
          (else (gro (cdr list) item  (cons (car list) list2))))))
(gro '(x y x z x w) 'x '())
    (w z y)
```

In reverse order! Could you put in original order?

Functions

- Functions are first class citizens in Scheme
  - Variables may be bound to functions
  - Can be passed as parameters
  - Can be returned as values of functions

Functions as First Class Citizens

A function may be bound to a variable, we are already doing this:

```
(define add1 (lambda(x) (+ 1 x)))

(add1 4)
5
```

Functions as First Class Citizens

Functions can be passed as parameters

```
(define (foo x y)
     (x y)
)

(foo (lambda(x) (+ 1 x)) 4)
5
```

Functions as First Class Citizens

Functions can be returned as values of functions

```
(define (foo x)
        (lambda(y) (+ x y)))
(foo 4)
#<procedure:2:16>

((foo 4) 5)
9
```

Examples (map)

```
(map cdr ‘((1 2) (3 4))
((2) (4))
(map car '((a b) (c d) (e f) (g h)))
(a c e g)
```

- Takes two arguments
  - Function and a list
  - Applies function to the list

Examples
(map)

```
(let
    ((proc (lambda(ls) (cons 'a ls))))
  (map proc '((b c) (d e) (f g h))))
```

((a b c) (a d e) (a f g h))

Apply

(apply + ‘(4 11))

15

(apply max ‘(3 4 5))

5

(apply <function> <arguments-in-a-list>)

Useful when the arguments are built separately from application

Eval

- Eval will evaluate the parameter as a valid Scheme expression

```
(eval ‘(car ‘(a b c))
    a
(eval ‘(define (foo a b) (+ a b)))
    foo
(foo 3 4)
    7
```

Allows for interesting opportunities for code to modify itself and execute self-generated code

Binding of Variables

- Global binding
  - define
- Local binding
  - lambda, let
- How do we change the value of a variable to which it is bound?
  - We have been using define multiple times, although in some implementations of Scheme this is invalid

set!

(set! var val)

- Evaluate val and bind it to var
- ! Indicates a side effect

- Scheme does not specify what this returns
  - Implementation dependent
  - Racket seems to return nothing

Examples

```
(define f (lambda(x) (+ x 10)))
(f 5)
>15
(set! f (lambda(x) (* x 10)))
(f 5)
>50
```

Examples

```
(let ((f (lambda(x) (+ x 100))))
  (display (f 5))
  (newline)
  (set! f (lambda(x) (* x 100)))
  (f 5))
```

- 105
- 500

(f 5)

- Error, reference to undefined identifier: f

set-car! & set-cdr!

(define x '(4 5 6))

(set-car! x 7)

x

> (7 5 6)

(set-cdr! x '(7 8 9))

x

>(7 7 8 9)

More Examples

```
(define (my-reverse ls)
   (cond ((null? ls) '())
         (else (append (my-reverse (cdr ls))
                       (list (car ls))))))

(my-reverse '(a b c d))
```

Recursive Functions

```
(define (atom? x) (not (list? x)))
(define (super-reverse ls)
   (cond ((null? ls) '())
         ((atom? ls) ls)
         (else (append (super-reverse (cdr ls))
                       (list (super-reverse (car ls)))))))

(super-reverse '((a b) ((c d) e) f))
```

Recursive Functions

```
(define (pairup x y)
   (cond ((null? x) '())
         ((null? y) '())
         (else (cons (list (car x) (car y))
                     (pairup (cdr x) (cdr y))))))

(pairup '(a b c) '(1 2 3))
```

Recursive Functions

```
(define (listify ls)
   (cond ((null? ls) '())
         (else (cons (list (car ls))
                     (listify (cdr ls))))))

(listify '(1 2 2 3))
```

Summary

- Pure functional programming
- No assignments (side effects)
- Refreshingly simple
- Surprisingly powerful
  - Recursion
  - Functions as first class objects
- Implicit storage management
  - Garbage Collection