CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 9

Recursive predicates, *continued*

Checking whether a symbol is a member of a list of symbols

```
(define member?
  (lambda (item lst)
    ...?...))


> (member? 'a '())
#f

> (member? 'a '(z a z))
#t

> (member? 'a '(a z a))
#t

> (member? 'a '(x y z))
#f
```

Recursive definition of `member?`

```
(define member?
  (lambda (item lst)
    (cond
      ;; Base case 1:
      ;; Reached the end; didn't find it.
      ((null? lst)
       #f)

      ;; Base case 2: Found it!
      ((equal? item (first lst))
       #t)

      ;; Recursive case: Keep looking.
      (else
       (member? item (rest lst))))))
```

Recursive definition of **member?**

```
(define member?
  (lambda (item lst)
    ;; ITEM is in LST if...
    (and (not (null? lst))
         (or (equal? item (first lst))
             (member? item (rest lst))))))
```

The built-in `member` function is like this, except it returns its input when it finds a match:

```
> (member 'b '(a b c))
(b c)
```

Why `(b c)` instead of `(a b c)`?

- `(b c)` is the input to the recursive call to member when it finds a match.

Checking whether a list of numbers is increasing

```
(define increasing?
  (lambda (lst)
    ...?...))
```

```
> (increasing? '())
#t
> (increasing? '(7))
#t
> (increasing? '(5 8 11 54))
#t
> (increasing? '(1 2 3 2 1))
#f
> (increasing? '(1 2 3 3 4 5))
#f
```

This is a “for all” question, unlike the “exists” question of `member?`.

- It’s trivially true when the list is empty.
  - Is it true that all Martians love their children? Sure, because there are no Martians.
- We want to return `#t` unless we can find a counterexample.

Constructing solution to larger problem from solution to smaller one

Argument = (x y_1 y_2 y_3 y_4 … y_n)

Case 1: $x < y_1$

Return: #t if (y_1 y_2 y_3 y_4 … y_n) is also increasing, otherwise return #f.

Case 2: $x \geq y_1$ – a *counterexample*!

Return: #f

Recursive definition of `increasing?`

```
(define increasing?
  (lambda (lst)
    (cond
      ;; Base case 1:
      ;;   No elements left; trivially true.
      ((null? lst) #t)

      ;; Base case 2:
      ;;   One element left; trivially true.
      ((null? (rest lst)) #t)

      ;; Recursive case:
      ;;   The first two elements are increasing;
      ;;   keep checking.
       ((< (first lst) (second lst))
        (increasing? (rest lst)))

      ;; Base case 3:
      ;;   Found a counterexample.
      (else #f))))
```

Recursive definition of `increasing?`

```
(define increasing?
  (lambda (lst)
    (or (null? lst)
        (null? (rest lst))
        (and (< (first lst) (second lst))
             (increasing? (rest lst))))))
```

Under the hood

You may have been wondering…

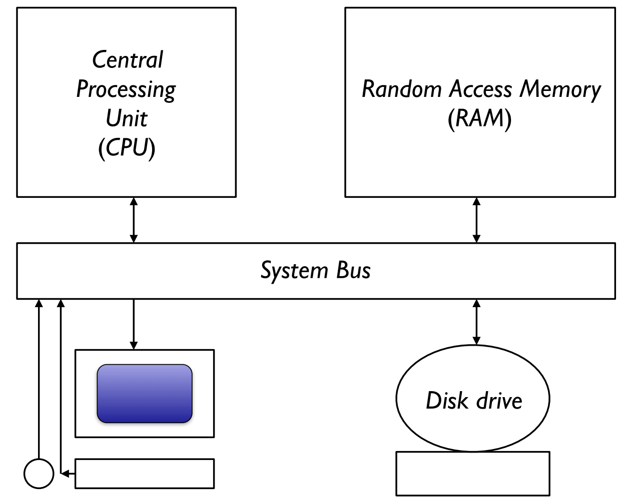
You've heard of bits and bytes.

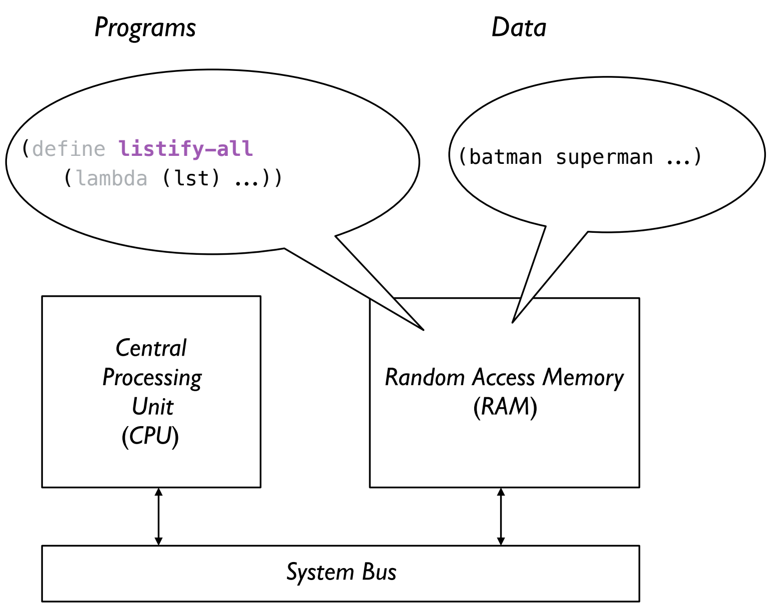
We've been talking about symbols and lists.

What's the connection?

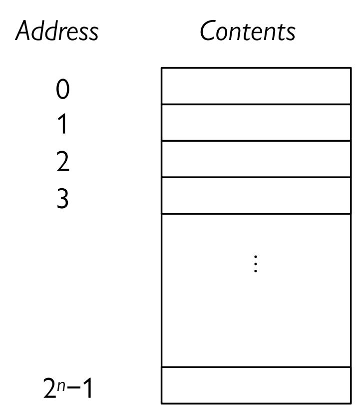
How are lists actually stored in the memory of a computer?

Von Neumann architecture





Organization of RAM



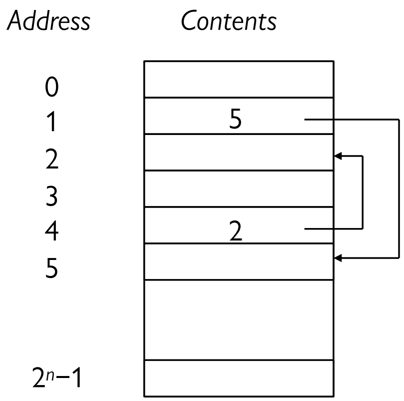
Memory is divided into storage locations.

Each location holds some data called its "contents".

Each location has a label called its "address".

Given the address, one can use it to find the location and get the contents.

Organization of RAM



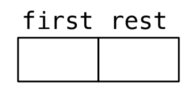
The address of one location may be stored as the contents of another.

The first location is said to “point” to the second.

We represent a pointer visually with an arrow from one location to another.

`cons`, **`first`**, and **`rest`**

A `cons` cell:

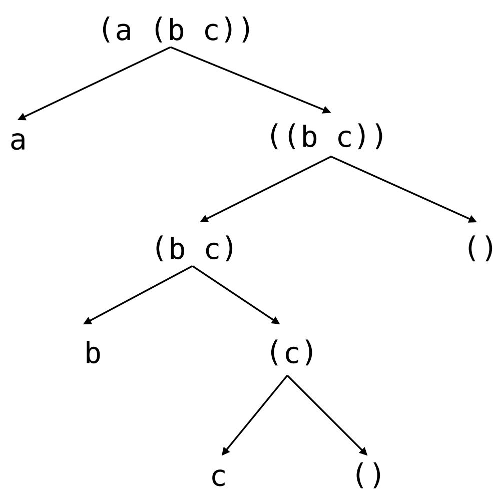


A block of RAM is divided into two parts:

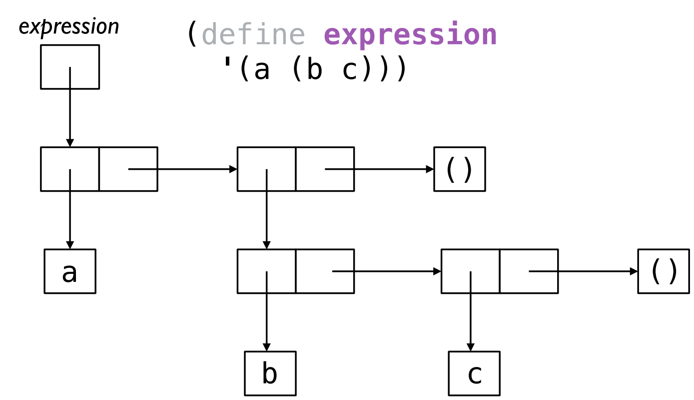
One part is the “`first`”.

The other part is the “`rest`”.

Splitting **`(a (b c))`** into **`firsts`** & **`rests`**



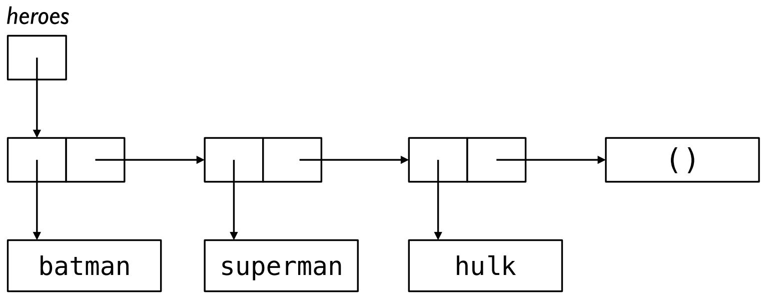
`cons` cell representation of **`(a (b c))`**




```
(define expression
  '(a (b c)))
```

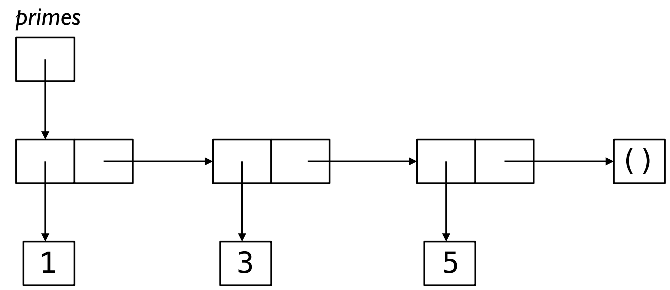
Representation of lists

```
(define heroes '(batman superman hulk))
```



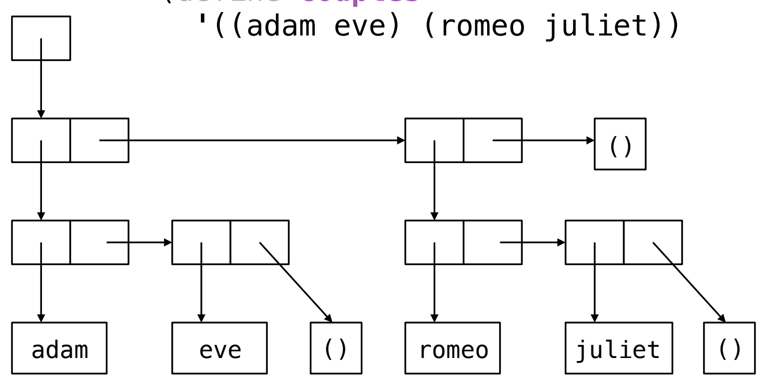
Representation of lists

```
(define primes '(1 3 5))
```



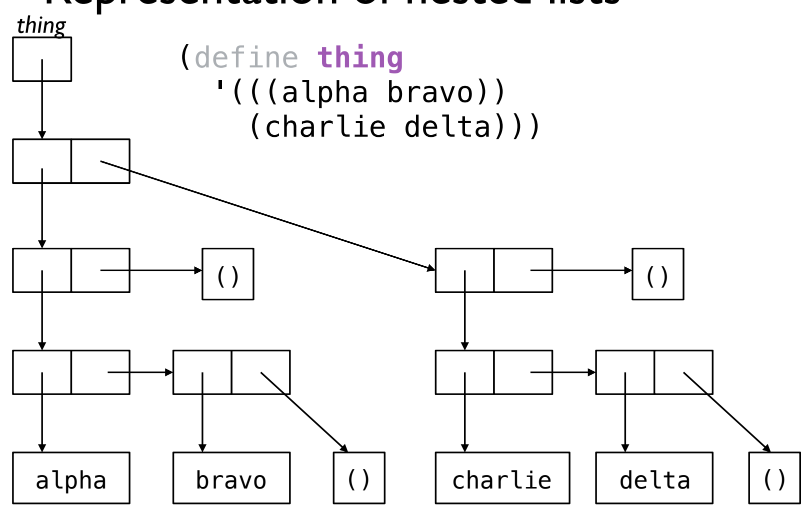
Representation of nested lists

```
(define couples
  '((adam eve) (romeo juliet))
```



Representation of nested lists

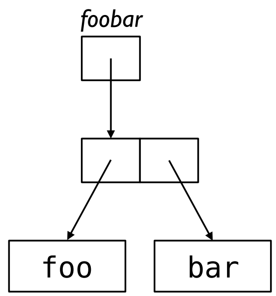
```
(define thing
  '(((alpha bravo))
    (charlie delta)))
```



What is this?

```
> (define foobar (cons 'foo 'bar))
> foobar
'(foo . bar)
```

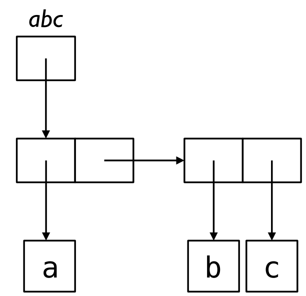
A “dotted pair”.



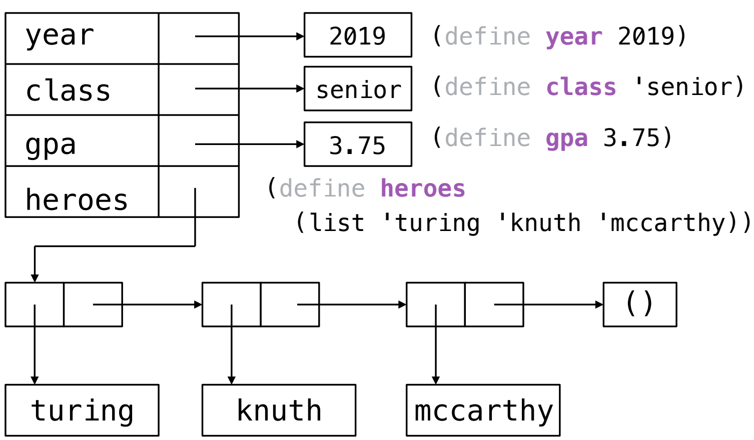
What is this?

```
> (define abc (cons 'a (cons 'b 'c)))
> abc
'(a b . c)
```

An “improper list”.



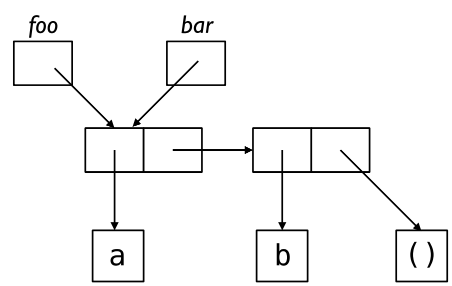
Scheme’s symbol table



Aliasing

```
(define foo '(a b))
(define bar foo)
```

We say that `bar` is *aliasing* `foo` because `foo` and `bar` refer to the same structure in memory.



Copying a list

```
(define copy
  (lambda (x)
    ...?...))


> (copy '(a b c))
'(a b c)

> (copy 'a)
'a
```

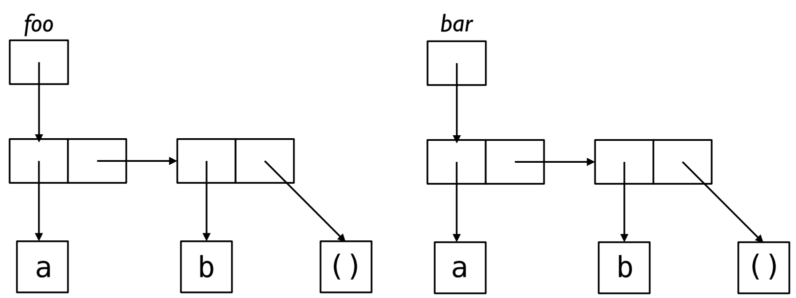
Copying a list

```
(define copy
  (lambda (lst)
    (if (not (pair? lst))
        lst
        (cons (copy (first lst))
              (copy (rest lst))))))
```

Illustration of copying

```
(define foo '(a b))

(define bar (copy foo))
```



Versions of equality

```
> (define foo '(a b))

> (define bar foo)

> (equal? foo bar)
#t

> (eq? foo bar)
#t
```

```
> (define foo '(a b))

> (define bar
    (copy foo))

> (equal? foo bar)
#t

> (eq? foo bar)
#f
```

Comparing **equal?** and **eq?**

The `equal?` procedure tests whether two lists are structurally equivalent.

- `equal?` takes more time for longer lists.

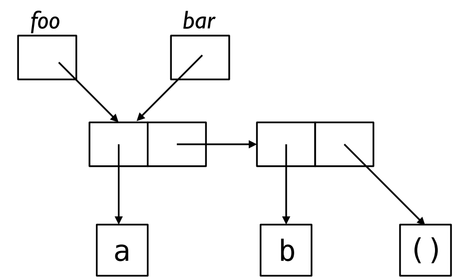
The `eq?` procedure tests whether two lists are the same object in memory.

- `eq?` takes the same time for all lists.

Equality and aliasing

```
(define foo '(a b))
(define bar foo)
```

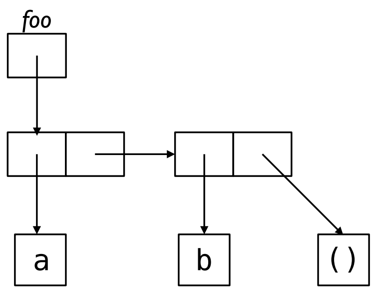
```
(equal? foo bar) → #t
(eq? foo bar) → #t
```

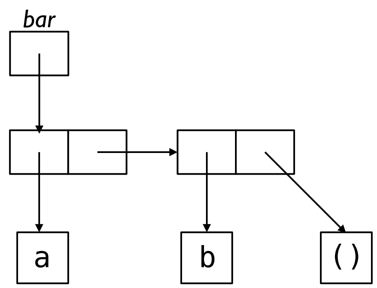


Equality and copying

```
(define foo '(a b))
(define bar (copy foo))
```

```
(equal? foo bar) → #t
(eq? foo bar) → #f
```





Acknowledgments

This lecture incorporates material from:

- Tom Ellman