LISP
Programming Paradigm

LISP (Scheme)

Scheme as a new LISP-like language

The language was conceived by **John McCarthy**

(Based on his paper *"Recursive Functions of Symbolic Expressions and Their Computation by Machine"* in 1960).

In the mid 70's **Sussman and Steele** (MIT ie Massachusetts Institute of Technology, USA) defined ***Scheme*** as a new LISP-like Language

Scheme has mostly been used as a language for teaching Computer programming concepts where as **Common** Lisp is widely used as a practical language

The `define` Function

- Used to define global "variables"; e.g.

  `(define f 120)`

  - **`define`** changes its environment but is not equivalent to an assignment statement. It just gives a name to a value.
  - **`define`** can also be used for other purposes, as we will see.

- ***`setQ`*** *and* ***`setF`*** *are functions that operate more like assignments; also* ***`set!`***

```
(define  X  2)

X
2

(* 5 X)
10
```

```
(define pi 3.14159)
(define radius 10)

(* pi (* radius radius))
314.159
```

Expressions

Cambridge prefix notation for *all* Scheme expressions:

(f x1 x2 … xn)

e.g.,

```
(+ 2 2)      ; evaluates to 4
(+(* 5 4 2)(-6 2));means 5*4*2+(6-2)
```

Note: Scheme comments begin with **;**

Cambridge prefix allows operators to have an arbitrary number of arguments.

Expression Evaluation

Three steps:

1. replace names of symbols by their current bindings.

2. Evaluate lists as function calls in Cambridge prefix, where a list is a set of elements enclosed in ( );

   e.g., **`(* a 2)`**

   The first element in the list is always treated as the function unless you specifically say not to.

3. Constants evaluate to themselves.

Expressions cont…

- An expression can take arbitrary number of arguments

```
(+ 21 35 12 7)
75

(* 25 4 12)
1200
```

- Nested Expression

```
(+ (* 3 5) (- 10 6))
19
```

- There is no limit (in principle) to the depth of such nesting

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
57
```

Lists as Function Calls

```
(+ 5 2 9 13)    ; evaluates to 29
```

but

```
(+ (5 2 9 13)); an error
```

Scheme will try to evaluate the second list, interpreting "5" as a function

```
(f); error - f isn't a function
```

Preventing Evaluation

```
(define colors (quote (red yellow green)))
  or
(define colors (` (red yellow green)))
```

Quoting tells Scheme/LISP that the following list is not to be evaluated.

Quoted Lists

```
(define x f)
  ; defines x as 120 (value of f)
(define x `f)
  ; defines x as the symbol f
(define color `red)
  ; color is defined to be red
(define color red)
  ; error: no definition of red
```

Defining Functions

- **`define`** is also used to define functions; according to the following syntax:

  `(define name (lambda (arguments) body))`

  or

  `(define (name arguments) body)`

  - From the former, one can see that Scheme is an applied lambda calculus.

<u>Example</u>

`(define (min x y) (if (< x y) x y))`

to return the minimum of x and y

Function

The general form of a function definition is

(define (<name> <formal parameters>) <body>)

```
(define (square x) (* x x))
```

Use the procedure *square*

```
(square 21)
441

(square (+ 2 5))
49

(square (square 3))
81
```

Function cont…

use **square** as a building block in defining other procedures

```
(define  (sum-of-squares x y)  (+ (square x) (square y)))

(sum-of-squares 3 4)
25
```

Now we can use **sum-of-squares** as a building block in constructing further procedures:

```
(define (plus1 x) (+ x 1))
(define (multby2  x) (* x 2))
(define (f a) (sum-of-squares (plus1 a) (multby2 a)))
```

or

```
(define (f a) (sum-of-squares (+ a 1) (* a 2)))

(f 5)
136
```

The Substitution Model for Function Application

```
(define (square x) (* x x))
(define (sum_of_squares x y)  (+ (square x) (square y)))
(define (f a) (sum_of_squares (+ a 1) (* a 2)))
(f 5)

Trace the evaluation
(f 5)
136



(f   5)
(sum_of_square  (+ 5 1) (* 5 2))
(+  (square (+ 5 1)) (square (* 5 2)))
(+  (*  (+ 5 1) (+ 5 1)) (*  (+ 5 2) (+ 5 2)) )
.
.
.
136
```

The Substitution Model for Function Application

Order of Evaluation

Applicative
(Evaluate the arguments and then apply)

```
(f   5)
(sum _of _square  (+ 5 1) (* 5 2))
(+  (square 6) (square  10))
(+  (*  6 6) ( * 10 10))
(+  36  100)
136
```

Normal
(Fully expand and then evaluate)

```
(f   5)
(sum _of _square  (+ 5 1) (* 5 2))
(+  (square (+ 5 1)) (square (* 5 2)))
(+  (* (+ 5 1) (+ 5 1)) (* (+ 5 2) (+ 5 2)))
.
.
136
```

Control Flow / Conditional statement

Conditional Statement

```
(cond (<p1> <e1>) (<p2> <e2>)......(<pn> <en>))
```

```
(cond (<p1> <e1>) (<p2> <e2>)......(else <en>))
```

```
(if <p> <e1> <e2>)
```

Consider a function

```
|x|     = 0  if x=0
        = x  if x>0
        = -x if x<0
```

In Scheme:

```
(define (abs x) (cond ((> x 0) x) ((= x 0) 0)((< x 0) (* -1 x))))
```

```
(define (abs x) (cond ((< x 0) -x)) (else x))
```

```
(define (abs x) (if  (< x 0) (- x) x))
```

Conditional statement

An example

```
(define (weather f) (cond  ((> f 38) 'too-hot)
                           ((> f 25) 'nice)
                           ((< f 10) 'too-cold)
                           (else  'moderate)))

(weather  10)

(weather  23)

(weather  41)
```

Logical Operators

```
LOGICAL AND :      (and <e1> ... <en>)
LOGICAL OR  :      (or <e1> ... <en>)
LOGICAL NOT :      (not <e>)
```

Consider an expression :

```
(and (> x 5) (< x 10))

(define (myfun a b)(if (and (> b a) (< b (* a b)))  b a))
```

Switch/Case

The **`case`** statement is similar to a Java or C++ **`switch`** statement:

```
(case month
   (( sep apr jun nov) 30)
   ((feb) 28)
   (else 31)    ; optional
   )
```

All cases take an unquoted list of constants, except for the `else`.

Recursion

```
Factoral(n) = 1                       if n=0
            = n*Factoral(n-1)  if n>0

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

```
Fibonacci(n)  = 0                    if n=1
              = 2                    if n=2
              = Fibonacci(n-1)+Fibonacci(n-2)   if n>2

(define (fib n)(cond ((=  n 1) 0)
                     ((=  n 2) 1)
                     (else (+ (fib (- n 1))(fib (- n 2))))))
```

Arithmetic expression

- **(exp x) -** which returns the value of e^x
- **(log x) -** which returns the value of the natural logarithm of x
- **(sin x) -** which returns the value of the sine of x **(cos x)(tan x)**
- **(sqrt x) -** which returns the principle square root of x
- **(max x_1 x_2...) -** which returns the largest number from the list of given numbers **(min x_1 x_2...)**
- **(quotient x_1 x_2) -** which returns the quotient of x_1 x_2.
- **(remainder x_1 x_2) -** which returns the integer remainder of x_1 x_2
- **(modulo x_1 x_2) -** returns x_1 modulo x_2
- **(gcd num1 num2 ...) -** which returns the greatest common divider from the list of given numbers
- **(lcm num1 num2 ...) -** which returns the least common multiple from the list of given numbers
- **(expt base power) -** which returns the value of base raised to power

Lists

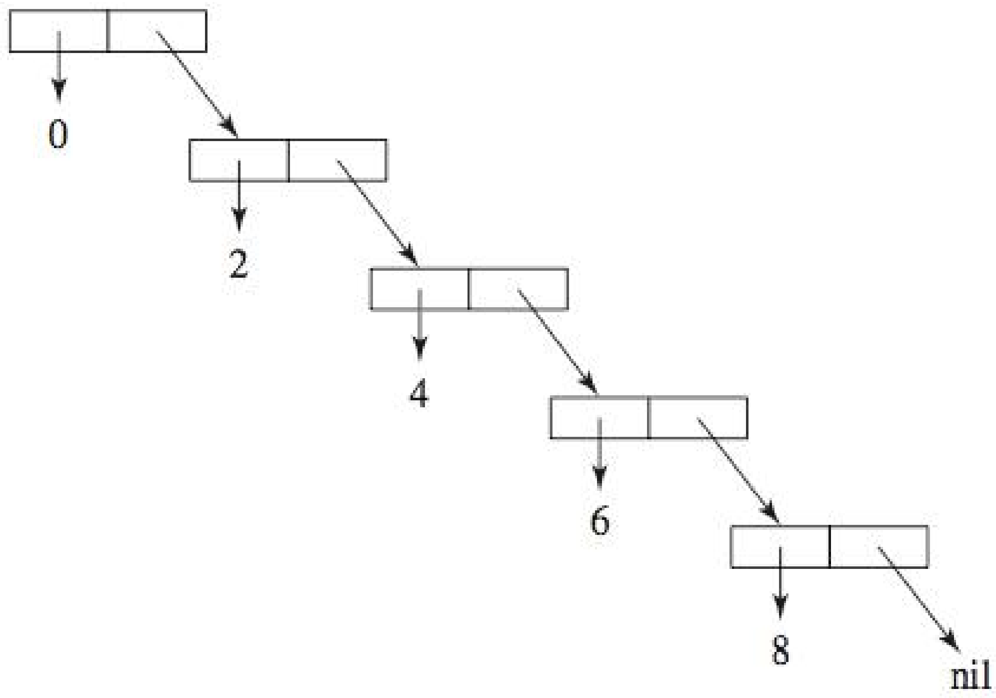
- A *list* is a series of expressions enclosed in parentheses.

  Lists represent both functions and data.

  The empty list is written *()*.

  e.g., *(0 2 4 6 8)* is a list of even numbers.

  Here's how it's stored:



List Node Structure

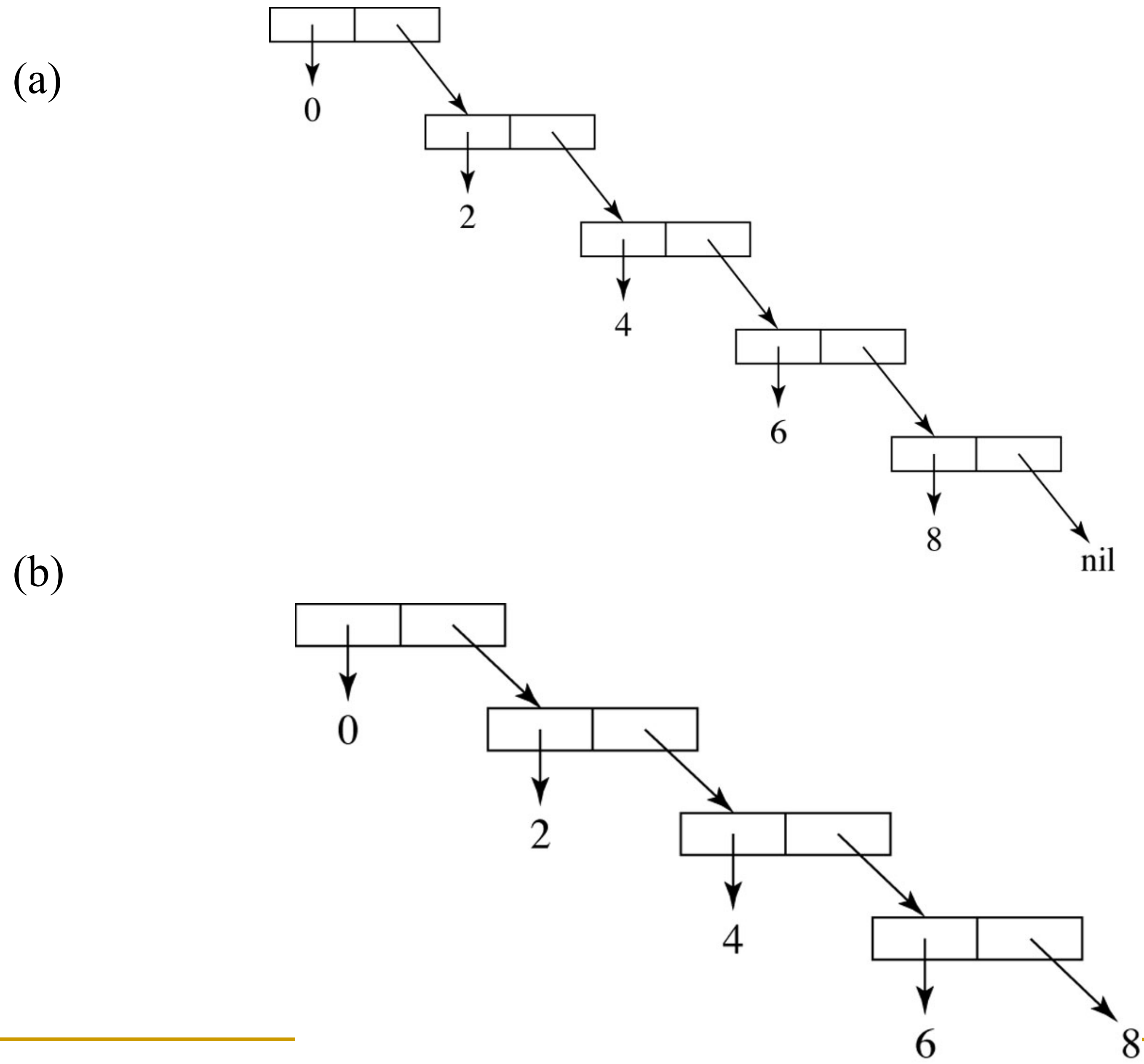
- Each list node is a record with two fields: the *car* and the *cdr: car* is (a pointer to) the first field, *cdr* is (a pointer to) the second.
- Note that in the previous example the *cdr* of the last node (|8|nil|) is *nil*.
  - This is equivalent to the null pointer in C/C++
- The nil value can be represented as ( ), which is also the representation for an empty list.

“Proper” Lists & Dotted Lists

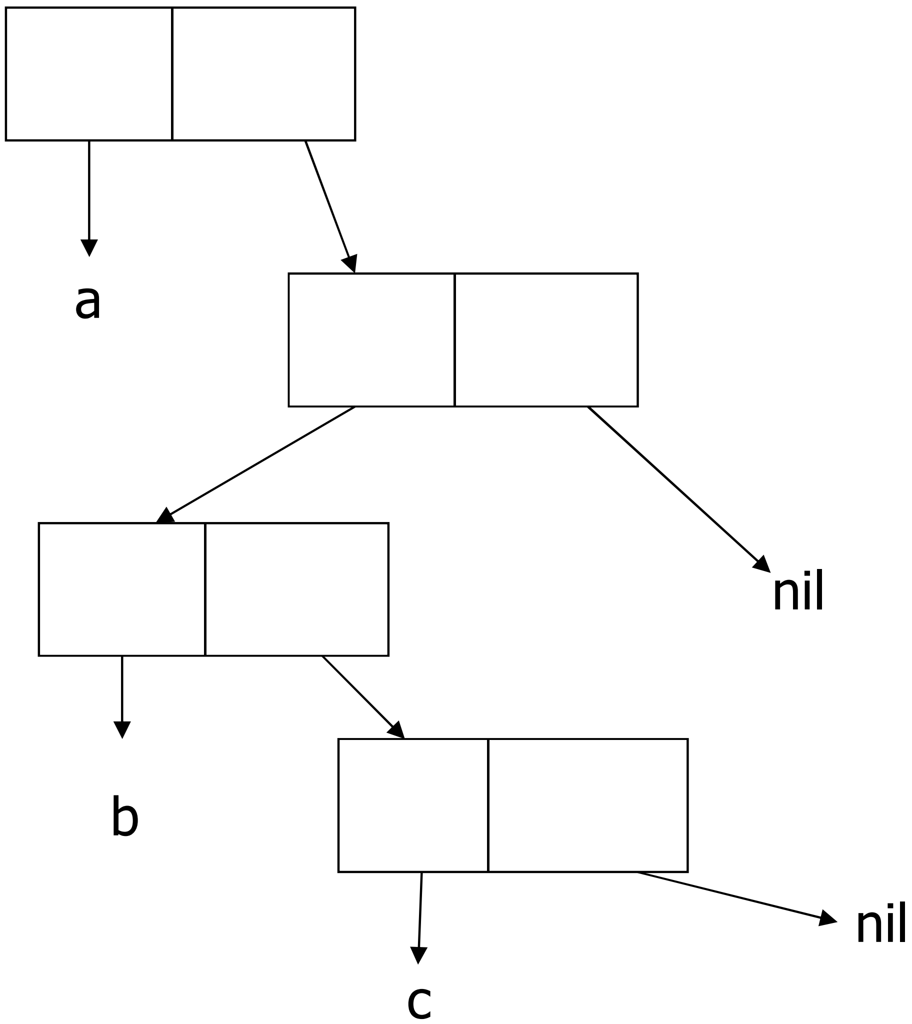
- Proper lists are assumed to end with the value ( ) which is implemented by null reference
- In a dotted list the last cons has some value other than nil as the value of the cdr field.
  - “Dotted” lists are written (0 2 4 6 . 8) The last node in the previous list would have been |6|8|

Structure of a List in Scheme

Notice the difference between the list with nil as the last entry, and the "dotted" list shown in part b.



List Elements Can Be Lists Themselves



This represents the list (a (b c))

List Functions

car: returns the first element (“head”) of a list

cdr: returns the tail of the list, which is itself a list

cons: used to build lists

List Transforming Functions

Suppose we write

```
(define evens `(0 2 4 6 8)).
```

Then:

```
(car evens)     ; gives 0
(cdr evens)     ; gives (2 4 6 8)
( (null? `())   ; gives #t, or true
(cdr(cdr evens)); (4 6 8)
(car`(6 8)) ; 6 (quoted to stop eval)
```

The *cons* requires two arguments: an element and a list; e.g.,

```
(cons 8 ( ))      ; gives the 1-element list (8)
(cons 6 (cons 8( ))) ; gives the list (6 8)
(cons 6 `(8))   ; also gives the list (6 8)
(cons 4(cons 8 9)) ; gives  the dotted list
                   ; (4  8 . 9 ) since 9 is not a list
```