CS61A NOTE13 Scheme Pairs and Lists

Pairs（类似 link list 只能两项） and Lists

All lists in Scheme are linked lists. Scheme lists are composed of two element pairs. We define a list as being either

- the empty list, `nil`　　空 list 是 nil
- a pair whose second element is a list 对的第二个元素是 list

As in Python, linked lists are recursive data structures. The base case is the empty list.

创建 list

We use the following procedures to construct and select from lists:

- `(cons first rest)` constructs a list with the given first element and rest of the list. For now, if `rest` is not a pair or `nil` it will error.用首个和剩下创建一个 list，剩余必须是 nil 或 pair
- `(car lst)` gets the first item of the list 首个
- `(cdr lst)` gets the rest of the list 后面

Scheme Lists - cons

There are three ways to create lists in Scheme:

`(cons <first> <rest>)`

- where **`<first>`** is an expression and **`<rest>`** is another Scheme list
- cons differs from our Link class in Python with the fact that you must explicitly pass in a **`<rest>`**: if you want a single element list, you should pass **`nil`** for **`<rest>`**
- commonly used for recursively creating a list where the **`<rest>`** is a recursive call expression to the function.

纯文本

```
scm> nil
() #空也是有()
scm> (define lst (cons 1 (cons 2 (cons 3 nil))))
lst
scm> lst
(1 2 3)   #要带()
scm> (car lst)
1
scm> (cdr lst)
(2 3)
```

Scheme lists are displayed in a similar way to the Link class we defined in Python.

Two other ways of creating lists are using the built-in `list` procedure or the `quote` special form.创建列表的另两种方式是使用内置 list 或引用

The `list` procedure has the syntax `(list <item> ...)`. It takes in an arbitrary number of operands and constructs a list with their values.

纯文本

```
scm> (list 1 2 3)
(1 2 3)
```

The `quote` special form has the syntax `(quote <expression>)`. It returns the literal `expression` without evaluating it. A shorthand for the `quote` special form is `'<expression>`.返回文字而非计算，缩写'<>

纯文本

```
scm> (define a 61)
```

```
a
scm> a
61
scm> (quote a)
a #直接返回本身
scm> 'a
a #返回文字
```

We can use the `quote` form to create a list by passing in a combination as the `expression`:

```
scm> (quote (1 x 3))
(1 x 3) #带()
scm> '(1 x 3)   ; Equivalent to the previous quote expression
(1 x 3) #引用不计算
```

An important difference between `list` (along with `cons`) and `quote` is that `list` and `cons` evaluate each of their operands before putting them into a list, while `quote` will return the list exactly as typed, without evaluating any of the individual elements.区别：list（以及 cons）和 quote 不同，list 和 cons 在把 operands 放进 list 之前会计算，quote 会完全按输入返回列表，不计算任何元素。

```
scm> (define a 1)
a
scm> (define b 2)
b
scm> (list a b 3)
(1 2 3) #内置list方法
scm> '(a b 3)
(a b 3) #quote方法，原样输出，带括号
```

Note that if we wanted to create the list `(a b 3)` using the `list` procedure, we could quote the symbols `a` and `b` so that they are not evaluated when making the list:想在 list 里仍然不计算，用引用

```
scm> (list 'a 'b 3)
(a b 3)
```

=, eq?, equal?

- `(= <a> <b>)` returns true if `a` equals `b`. Both must be numbers.比较数字（数字==）
- `(eq? <a> <b>)` returns true if `a` and `b` are equivalent primitive values. For two objects, `eq?` returns true if both refer to the same object in memory. Similar to checking identity between two objects using `is` in Python 对两个对象，若两个引用同一个内存中的对象则 True （is）
- `( equal? <a> <b>)` returns true if `a` and `b` are *pairs* that have the same contents (`car` s and `cdr` s are equivalent)对内容相同. Similar to checking equality between two lists using `==` in Python. If `a` and `b` are not pairs, `equal?` behaves like `eq?`. 用来比较 pair 的，不是 pairs 的话与 eq 一致（一切==）

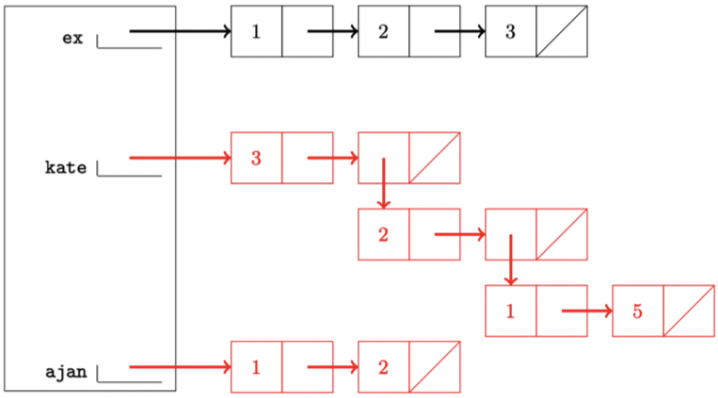
纯文本

```
scm> (define a '(1 2 3))
a
scm> (= a a)
Error       #不是数字不能比
scm> (equal? a '(1 2 3))
#t          #是pair，内容相同
scm> (eq? a '(1 2 3))
#f          #对两个对象，若两个引用同一个内存中的对象则True
```

Q1: Box and Pointers (Su18 Final #3b) Solution

(b) (3 pt) Draw a box-and-pointer diagram for the state of the Scheme pairs after executing the block of code below. Please erase or cross out any boxes or pointers that are not part of a final diagram. This code does not error. We've provided the diagram for **ex** as an example. The built-in procedure **length** returns the length of a Scheme list.

```
1  (define ex '(1 2 3))
2
3  (define (f x)
4    (if (= x 0)
5        5
6        (list x (f (- x 1)))))
7
8  (define kate (f 3))
9
10 (define (g x)
11   (if (list? x) (length x) x))
12
13 (define ajan (map g '(1 (2 (3)))))
```



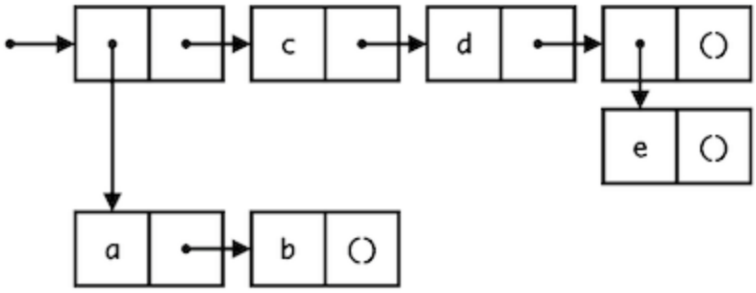
map g '（1(2(3)))，是 list，长度 1

WWSD

```
纯文本
scm> (cons 1 (cons 2 nil)) #注意后面写nil
(1 2)  #没有',' cons建立
scm> (car (cons 1 (cons 2 nil)))
1   #scheme的list是link list因此前面不带()
scm> (cdr (cons 1 (cons 2 nil)))
(2) #scheme的list是link list因此后面是()

scm> (list 1 2 3)
(1 2 3)

scm> '(1 2 3)
(1 2 3)

scm> (cons 1 '(list 2 3))
(1 list 2 3) #⚠️ cons只能联合两项，但不意味只有两格

(cons 1 '(2 (cons 3 nil)))
(1 2 (cons 3 nil))

scm> '(cons 4 (cons (cons 6 8) ()))
(cons 4 (cons (cons 6 8) ()))

scm> (cons 1 (list (cons 3 nil) 4 5)) #(cons 3 nil)是(3),
后面类似cons1 nil
(1 (3) 4 5)
```

Q2: List Making

用 list,quote,cons 写链表



纯文本

```
#((a,b),c,d,(e))
#用list
(define with-list    #写法参考(list 'a 'b 3)字符用' 无逗号无括号,每个list加一个()
    (list
        (list 'a 'b) 'c 'd (list 'e)
    )
)


#用quote
(define with-quote
    '(
        (a b) c d (e)
    )
)


#用cons，参考(cons 'a (cons 2 nil))
(define helpful-list
   (cons 'a (cons 'b nil)))
(draw helpful-list)

(define another-helpful-list
    (cons 'c (cons 'd (cons (cons 'e nil) nil))))
(draw another-helpful-list)

(define with-cons
    (cons
        (cons 'a(cons 'b nil)) (cons 'c(cons 'd (cons(cons 'e nil) nil)))
    ) #⚠一个cons(a nil)出来a，cons(cons(a nil) nil)出来(a)
)
```

Q3: List Concatenation

Write a function which takes two lists and concatenates them 连接起来

Notice that simply calling 只用 `(cons a b)` would not work because it will create a deep list. Do not call the built-in procedure `append`, since it does the same thing as `list-concat` should do.只用（cons a b）实现连接

```
纯文本
(expect (list-concat '(1 2 3) '(2 3 4)) (1 2 3 2 3 4))
(expect (list-concat '(3) '(2 1 0)) (3 2 1 0))




(define (list-concat a b)      #递归，输入更少+定义初始
    (if (null? a)              #参考(if (< 4 5) 1 2)
    a
    (cons(car a) (list-concat (cdr a) b))  #cdr a和b都可以nil，但此处a不能是ni
    )
)

#关于car和cdr
scm> (car lst)
1
scm> (cdr lst)
(2 3)
```

#加括号情况：实现功能/判断条件

Q4: Remove

Implement a procedure `remove` that takes in a list and returns a new list with *all* instances of `item` removed from `lst`. You may assume the list will only consist of numbers and will not have nested lists.

```
纯文本
(expect (remove 3 nil) ())
(expect (remove 2 '(1 3 2)) (1 3))
(expect (remove 1 '(1 3 2)) (3 2))
(expect (remove 42 '(1 3 2)) (1 3 2))
(expect (remove 3 '(1 3 3 7)) (1 7))
```

```
(define (remove item lst)
  #法一递归，定义初始
  (cond ((null? lst) '())
        ((equal? (car lst) item) (remove item (cdr lst)))
        (else (cons (car lst) (remove item (cdr lst))))
  )
)

#用filter?
(define (remove item lst)
  (filter (lambda (x) (not (= x item))) lst)
)
```

Q5: List Duplicator

Write a Scheme function, `duplicate` that, when given a list, such as `(1 2 3 4)`, duplicates every element in the list (i.e. `(1 1 2 2 3 3 4 4)`).

纯文本

```
(expect (duplicate '(1 2 3)) (1 1 2 2 3 3))
(expect (duplicate '(1 1)) (1 1 1 1))

(define (duplicate lst)
    (if (null? lst) #递归，定义初始
        '()
        (cons (car lst) (cons (car lst) (duplicate (cdr lst))))
    )
)
```

hw8 Q1: Pow

Implement a procedure `pow` for raising the number `base` to the power of a nonnegative integer `exp` for which the number of operations grows logarithmically, rather than linearly (the number of recursive calls should be much smaller than the input `exp`). For example, for `(pow 2 32)` should take 5 recursive calls rather than 32 recursive calls. Similarly, `(pow 2 64)` should take 6 recursive calls.基提高到非负整数 exp 的幂，对数增长

> *Hint:* Consider the following observations:
>
> 1. x2y = (xy)2
> 2. x2y+1 = x(xy)2
>
> For example we see that 232 is (216)2, 216 is (28)2, etc. You may use the built-in predicates `even?` and `odd?`. Scheme doesn't support iteration in the same manner as Python, so consider another way to solve this problem.

纯文本

```
(define (pow base exp)
    (cond
      ((= exp 0) 1)
      ((even? exp) (square(pow base (/ exp 2) )))
      (else (* base (pow base (- exp 1) )))
    )
)
```

hw8 Q2: Repeatedly Cube

Implement the following function, which cubes the given value `x` some number `n` times, based on the given skeleton.对 x 进行 n 次立方

Here are some examples of how `repeatedly-cube` should behave:

> scm> (repeatedly-cube 100 1) ; 1 cubed 100 times is still 1
>
> 1
>
> scm> (repeatedly-cube 2 2) ; (2^3)^3
>
> 512
>
> scm> (repeatedly-cube 3 2) ; ((2^3)^3)^3
>
> 134217728

纯文本

```
(define (repeatedly-cube n x)
  (if (zero? n)
      x
      (let(
            (y (repeatedly-cube (- n 1) x))
```

```
            )
          (* y y y)
        )
    )
)

参考
let (
      (x 5)
      (y 10)
     )
```

hw8 Q3: Thane of Cadr

Note: *Scheme lists will be covered in lecture on Wednesday, April 12. If you are working ahead, consider looking at the Scheme Specification for details on using* `car` *and* `cdr`*.*

Define the procedures `cadr` and `caddr`, which return the second and third elements of a list, respectively.

纯文本

```
(define (cddr s)
(cdr (cdr s))
)

(define (cadr s)
(car (cdr s))
)

(define (caddr s)
(car (cddr s))
)
```

hw9 Q1: Ascending

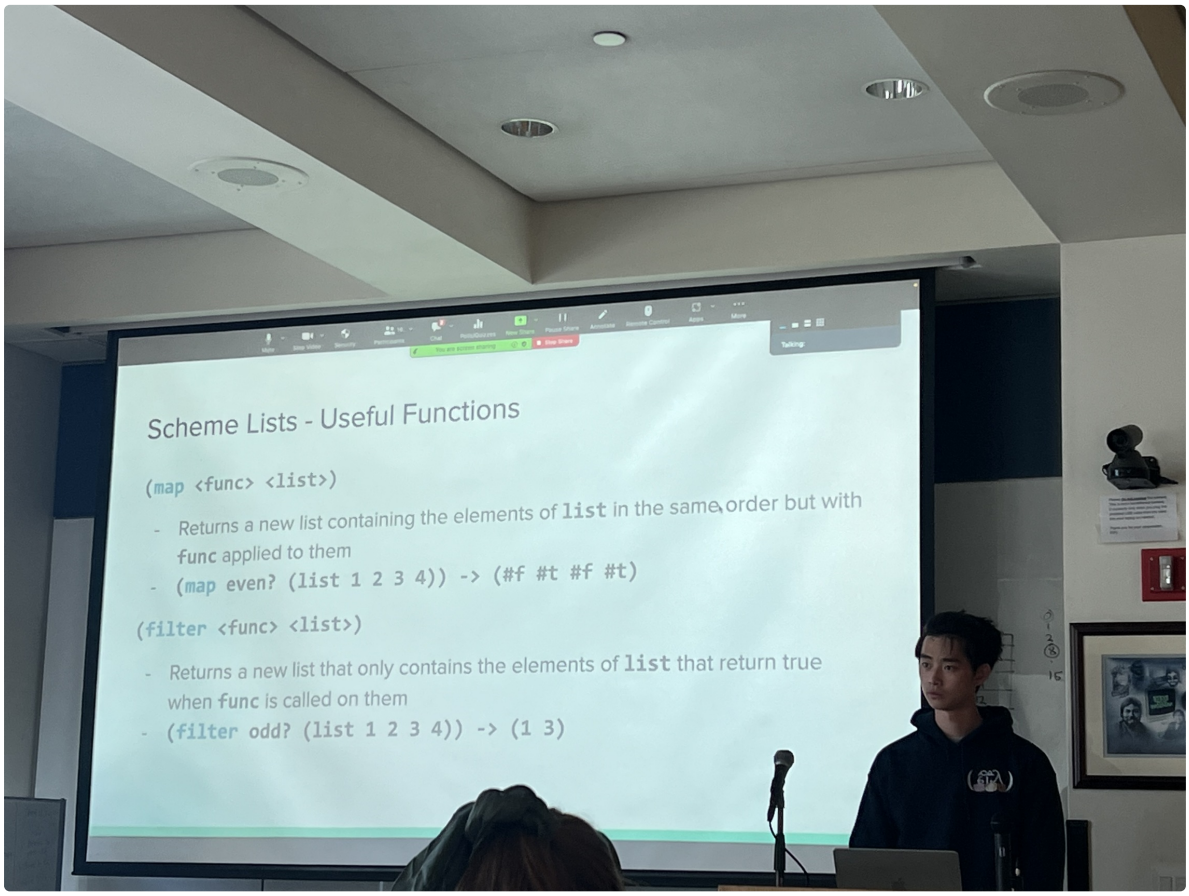
Implement a procedure called `ascending?`, which takes a list of numbers `asc-lst` and returns `True` if the numbers are in nondescending order, and `False` otherwise. Numbers are considered nondescending if each subsequent number is either larger or equal to the previous, that is:1 2 3 3 4 is ok

纯文本

```
(define (ascending? asc-lst)
```

```
  (if (or (null? asc-lst) (null? (cdr asc-lst)))
      #t
      (and (<= (car asc-lst) (car (cdr asc-lst))) (ascending? (cdr asc-lst)))
  )
)
```

hw9 Q2: My Filter 别的地方应用



Write a procedure `my-filter`, which takes a predicate `pred` and a list `s`, and returns a new list containing only elements of the list that satisfy the predicate. The output should contain the elements in the same order that they appeared in the original list.

Note: Make sure that you are not just calling the built-in `filter` function in Scheme – we are asking you to re-implement this!

纯文本

```
(define (my-filter pred s)
  (cond
        ((null? s) '())
        ((pred (car s)) (cons (car s) (my-filter pred (cdr s))))
        (else (my-filter pred (cdr s)))
```

```
  )
)
```

hw9 Q3: Interleave

Implement the function `interleave`, which takes a two lists `lst1` and `lst2` as arguments. `interleave` should return a new list that interleaves the elements of the two lists. (In other words, the resulting list should contain elements alternating between `lst1` and `lst2`.)

If one of the input lists to `interleave` is shorter than the other, then `interleave` should alternate elements from both lists until one list has no more elements, and then the remaining elements from the longer list should be added to the end of the new list.

纯文本

```
(define (interleave lst1 lst2)
  (cond
    ((null? lst1) lst2)
    ((null? lst2) lst1)
    (else (cons (car lst1) (interleave lst2 (cdr lst1))))
  )     #注意顺序
)
```

hw9 Q4: No Repeats

Implement `no-repeats`, which takes a list of numbers `lst` as input and returns a list that has all of the unique elements of `lst` in the order that they first appear, but no repeats. For example, `(no-repeats (list 5 4 5 4 2 2))` evaluates to `(5 4 2)`.

> **Hint:** How can you make the first time you see an element in the input list be the first and only time you see the element in the resulting list you return?

> **Hint:** You may find it helpful to use the `my-filter` procedure with a helper `lambda` function to use as a filter. To test if two numbers are equal, use the `=` procedure. To test if two numbers are not equal, use the `not` procedure in combination with `=`.

纯文本

```
(define (no-repeats lst)
```

```
  (if (null? lst)
    lst
    (cons (car.lst)
       (no-repeats (my-filter (lambda (x) (not (= (car lst) x))) (cdr lst)))
    )
   )
)
注意prep是条件/lambda
之前的key是max min sum/lambda
```

lab12 Q1: Substitute 替补

Write a procedure `substitute` that takes three arguments: a list `s`, an `old` word, and a `new` word. It returns a list with the elements of `s`, but with every occurrence of `old` replaced by `new`, even within sub-lists. 返回 list 旧的被替换成新的，即使是子列

Hint: The built-in `pair?` predicate returns True if its argument is a `cons` pair.

Hint: The `=` operator will only let you compare numbers, but using `equal?` or `eqv?` will let you compare symbols as well as numbers. 用 equal?

纯文本

```
(define (substitute s old new)
  (cond ((null? s) nil)
        ((pair? (car s))
          (cons (substitute (car s) old new) (substitute (cdr s) old new))
        )
        ((equal? (car s) old) (cons new (substitute (cdr s) old new)))
        (else (cons (car s) (substitute (cdr s) old new)))
  )
)
#
```