PPL Thing

`begin`

Used to sequence multiple expressions, evaluating them in order and returning the result of the last one. It is useful when you need to perform side effects, like printing or modifying state, before producing a final result.

```
(begin expr1 expr2 ... exprN)
```

- `expr1`, `expr2`, ..., `exprN` are evaluated in order.
- The result of `exprN` is returned

`car`

- Returns the **first** element of a pair (or the head of a list).
- "Contents of the Address Register"

`cdr`

- Returns the **rest** of the pair (or the tail of a list), which is everything except the first element.
- "Contents of the Decrement Register"

`cons` Function:

```
(cons a d) → list?
  a : any/c
  d : list?
(cons a d) → pair?
  a : any/c
  d : any/c
```

Returns a newly allocated *pair* whose first element is a and second element is d.

- When d is a list, the allocated pair is also a list.

`append` Function:

```
(append lst ...) → list?
  lst : list?
(append lst ... v) → any/c
  lst : list?
  v : any/c
```

When given all list arguments, the result is a list that contains all of the elements of the given lists in order. The last argument is used directly in the tail of the result.

Write a program in Scheme using "tail recursion" to compute the squares of all the elements in a list. Examples of inputs and outputs:

`(TR_sqr_list '(3 0 6)) → '(9 0 36)`
`(TR_sqr_list '(2 1 3 1)) → '(4 1 9 1)`

```
(define (TR_sqr_list l)
  (letrec
  ((helper (lambda (result l)
  (if (null? l)
     result
  (helper (append result (list (* (car l) (car l)))) (cdr l))))))
  (helper '() l)
  )
)
```

Use "delay" to create an infinite (lazy) list of all the cubic numbers

such as `1 (=1*1*1), 8 (= 2*2*2), 27 (=3*3*3), ...`

```
(define lazy_cube
 (letrec
    ((next (lambda (n) (cons (* n n n) (delay (next (+ n 1)))))))
  (next 1)))
```


C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	-9,223,372,036,854,775,808-9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807	8	9,223,372,036,854,775,808-9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

- There are several ways to make automatic garbage collection
  - Python and other languages automatically do this, but you can write your own scripts in C to do it
  - 4 Main Methods
    - **reference** $cunt$
      - Each thing on the heap has a reference count int, equal to the number of pointers pointing to it
      - If this drops to 0, then recycle it
      - An eager approach
        - If you're too eager, you'll sometimes have cycles of reference (a chain of references that are referencing them each other) this can lead to things that are unaccessible staying in the heap after they should've been deleted
    - **mark-and-sweep**
      - Mark each thing in the heap that has a reference, then recycle everything that's not marked.

      - Issue: Fragmentation: Your heap will be filled with scattered data, so if you need continuous space (for something like an array) you will have enough space but won't be able to fit it in there
    - **stop-and-copy**
      - divide it into two parts, only use one part at any given time, after clearing garbage, move all the useful stuff to the other part in continuous memory, now use this new, clean part and clear the old one to keep the cycle going
      - Advantage: It is able to reclaim garbage that contains cycles of references.
      - Disadvantage: Twice as much memory is needed for a given amount of heap space.
    - **generational**
      - Divide memory into a few generations
      - When you want some new memory, you only use open memory from one generation
      - When you run out of space in a generation, you move all the still useful stuff to the next generation
      - The Idea: The latest generation is mostly full with useful stuff with few gaps, while its likely that your oldest generation mostly has garbage

Scheme : functional programming paradigm

Use a code example in C to illustrate that a side effect makes a program hard to understand.

Your Answer:

```
#include <stdio.h>
int j = 12;
int f(int i)
{ j = j + 5;
return i + j;
}
int g(int a, int b)
{ return a - b; }
int main() {
printf("%d\n", g(f(3), f(3)));
printf("%d", f(2) - f(1));
```

Side effects: modifying a state variable outside its local scope.
referential transparency: if you call a function w/ same set of inputs multiple times, you'll always get same results.

(if #f (+ 9 8) (- 8 5))

3

3 (with margin: 0)

((if (null? '(+ -)) + -) 6 5)

1

1 (with margin: 0)

(car (cdr `(car (cdr (OU OSU)))))

→ (cdr (OU OSU))

Question 3

no ' throws error

cdr '(9 10 11) = (10 11)

(car (cdr '(9 10 11)))

10

Question 5

(+ (* 3 3) 5)

14

* ' means the list objects should be treated as literal data & not evaluated.
↑ precedence.

((if (< 3 4)
(lambda (x) (+ x 1))
(lambda (x) (* x 2))
)
6
)

7

(let ((x 3))
(let ((y x)) y = x = 3 b/c x=4 has not happened yet.
(squaresum (lambda (a b) (+ (* a a) (* b b
)
(squaresum x y)
4 3
)
)

local vars > global vars.

25

25 (with margin: 0)

Question 3

(eval (apply (lambda (x) (cdr x)) '((* + 3 4))))

7

(eval '(car '(4 5 6)))

4

4 (with margin: 0)

Question 5

(map (lambda (x) (* 3 x)) `(1 5 2))

Your Answer:

'(3 15 6)

→ cons syntax: (cons element1 element2)
ex: (define pair (cons 1 2))

→ apply syntax: (apply procedure arg1 arg2 ... argn list)
ex: (apply + 1 2 '(3 4 5)) ; prints 15

→ map syntax: (map procedure list1 list2 ...)
ex: (map - '(3 4 5) '(2 3 4)); prints (1 1 1)

→ lambda syntax. (lambda (param1 param2 ...) body)
ex: ((lambda (x) (+ x 2) 3) ; prints 5

→ let syntax:
(let ((binding1 expr1)(...)..) body)
ex: (let ((x 10)) (+ x 5)) ; prints 15

Write a (purely functional) Scheme function that multiplies two lists component-wisely, padding 1 if necessary.

Examples of inputs and outputs

(mlist '(1 2) '(3 4 5)) --> '(3 8 5)

Your Answer:

```
(define (mlist l1 l2)
(if (null? l1) l2
(if (null? l2) l1
(cons (* (car l1)(car l2)) (mlist (cdr l1)(cdr l2)))
)))
```

letrec : defines mutually recursive procedures. (mutual rec is where 2+ procedures call each other in their defs)
→ syntax: (letrec ((name1 lambda-expr1)(name2 lambda-expr2) ... (namen-lambda-exprn)) body-expr)
→ ex:

```
(define result
  (letrec ((even?
            (lambda (n)
              (if (= n 0)
                  #t
                  (odd? (- n 1)))))
           (odd?
            (lambda (n)
              (if (= n 0)
                  #f
                  (even? (- n 1))))))
    (even? 5)))

(display result) ; Output: #f
```

Write a program in Scheme using "tail recursion" to compute the scalar multiplication.
Examples of inputs and outputs:
(scalarmul '(1 2 0) 3) ---> '(3 6 0)
(scalarmul '(3 4 5) 10) ---> '(30 40 50)

Your Answer:

```
(define (scalarmul lst scalar)
 (define (helper lst scalar acc)
  (if (null? lst)
      (reverse acc)
      (helper (cdr lst) scalar (cons (* (car lst) scalar) acc))
  )
 )
 (helper lst scalar '())
)
```

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

Use "delay" to create an infinite (lazy) list of all the cubic numbers such as 1 (=1*1*1), 8 (= 2*2*2), 27 (=3*3*3), ...

Your Answer:

```
(define lazycubes (letrec ((next (lambda (n) (cons (* n (* n n)) (delay (next (+ n 1))))))) (next 1)))
```

body-expr

delay & force used to implement lazy eval (avoid unnecessary comps).
delay creates a promise
→ syntax: (delay expression)
→ ex: (define a (delay (+ 35)))
a ; prints <promise: a>
force delivers promise by evaluating
→ ex: (force a) ; prints 8.

```
(define lazycubes
  (letrec ((next (lambda (n)(cons (* n (* n n))(delay(next (+ n 1)))))))(next 1))

(define naturals
  (letrec ((next (lambda (n) (cons n (delay (+ n 1)))))) (next 1)))
```

python : multi-paradigm (OOP, FP) → LEGB

```
print((lambda x,y:x[y])("Texas", 3 ))
```

Your Answer:

a

```
x=[ 'Maine', 2]
def f():
  x[1]=[3,2]
print(x)
f()
print(x)
```

Your Answer:

['Maine', 2]
['Maine', [3,2]]

Question 2

```
y=7
x=[4,5]
if y==7:
  x = [7,8,9]
print(x[1])
```

precedence b/c LEGB

8

Question 4

```
l=[x+"T" for x in "twitter" if x != "t"]
print(l)
l.append(7)
l=[2*x for x in l]
print(l[-1])
```

Your Answer:

['wT', 'iT', 'eT', 'rT']
14

Use "list comprehension" to write a Python list that enumerates first 100 integers that can be written as sums of even integer squares and 1.
So the first four integers in the list should be 1 (= 0^2+1), 5 (= 2^2+1), 17 (= 4^2 + 1), 37 (= 6^2 + 1).

```
[(2**n)**2+1 for n in range(100)]
```

→ list comprehension
• for i in range(3) is just [0,1,2,3) (don't include 3).

```
for x in range(3):
  x = x+3
  print(x)
print(x)
for x in range(4):
  print(x)
```

1. 3
2. 4
3. 5
4. 5
5. 0

```
l = [1, 3, 'oklahoma', 3]
print(l[1:3])
# prints [3,'oklahoma']

(1*3)[2:-3]
# ['oklahoma', 3, 1, 3, 'oklahoma', 3, 1]

[i for i in range(100)]
# creates a list of elements 0 - 99

[c for c in 'oklahoma' * 3]
# creates a list. Each element is a character in 'oklahoma' and is added 3 times
# ['o', 'k', etc... 'o', 'k', etc]

print([[i for i in range(j)] for j in range(10)])
# prints [[], [0], [0, 1], [0, 1, 2], [0, 1, 2, 3], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4, 5], [0, 1, 2, 3, 4, 5, 6], [0, 1, 2, 3, 4, 5, 6, 7], [0, 1, 2, 3, 4, 5, 6, 7, 8]]

l = ["ohio", 8,"dallas", 0, 0, 0]
while l[-1] == 0:
    del l[-1]

print(l)
# prints ['ohio', 8, 'dallas']
```

```
x = (1, 3, "littlerock") ← tuple is immutable
print (x)
```

```
x = 3
def f():
    # print(x) taking this out of comment will generate unboundlocalerror (cannot access local variable 'x' where it is not associated with a value)
    x = 'houston'
    print(x)
```

C: imperative lang.

```
class University:
  def __init__(self, nm, lo, pr):
    self.nm = nm
    self.lo = lo
    self.pr = pr

  def __call__(self, cn, dean):
    return College(self.nm, self.lo, cn, dean)

class College:
  Purpose = "Professional Education"
  def __init__(self, un, lo, cn, dean):
    self.un = un
    self.lo = lo
    self.cn = cn
    self.dean = dean

um = University("U. of Missouri", "Ann Arbor", "Mary Cruz")
um.Purpose = "Research"
ocu = University("U. of North Oklahoma", "Edmond", "Alice Betz")
ocucoa = ocu("College of Fine Arts", "Henry Christie")
ocucoa.lo = "OKC"
```

can call like a function.

print(um.nm)

Your Answer:

U. of Missouri

Question 2

print(ocu.Purpose)

Your Answer:

[redacted] Error

Question 3

print(ocucoa.lo)

Your Answer:

OKC

Question 4

print(ocucoa.Purpose)

Your Answer:

Professional Education

print(um.Purpose)

Your Answer:

Research

python, R, JS, TS, perl

Scripting: sequences of instructions interpreted by computer rather than compiled into machine code.
① text processing: manipulate & transform textual data
② Extension languages: Scripting languages embedded within other software apps to to extend or customize their func.
③ Glue languages: connect & integrate different components or systems.

Benefits:
- automation
- customization
- integration
- rapid prototyping

yield & generators:
generator: special type of iterator that lets you iterate over large sequence of data w/o creating it all at once.
- When a function contains 'yield,' it becomes a generator function.

Use yield to write a Python generator sofsq that produces all the sums of squares. The first three sofsq are
1*1(= 1), 1*1+2*2(= 5), 1*1+2*2+3*3(= 14)
You should write only one function (generator), i.e. you should not use helper functions.

Your Answer:

```
def sofsq():
    n = 1
    sum = 0
    while True:
        sum += n * n
        yield sum
        n += 1
```

```
def degreeonepoly():
  poly = [0, 0]
  yield poly
  sign = -1
  n = 1
  while True:
    for i in range(n):
      poly[1] += sign
      yield poly
    for i in range(n):
      poly[0] += sign
      yield poly
    n += 1
    sign *= -1

for i in degreeonepoly():
  print(i)
```

```
def g():
        yield 1
        yield 2
        yield "Tulsa"
for i in g():
        print(i)
# prints:
# 1
# 2
# Tulsa
def squares():
        i = 0;
        while True:
                yield i*i
                i = i + 1
for s in squares():
        print(s)
        if s > 10000:
                break
```

```
#include <stdio.h>
int a=3;
int b;
b = a + (a=5);
printf("%d",b);
```

undefined

```
#include <stdio.h>

if (-3< 3 <2)  printf("%d\n", 35);
else printf("%d\n", 24);
```

35

35 (with margin: 0)

Question 3

```
#include <stdio.h>

int main() {
   int a = 3;
   int b = 0;

   if (a = b)
      printf("%d\n", 3323);
   else
      printf("%d\n", 101);
}
```

101

```
#include <stdio.h>
void f(int i);

int main()
{
  f(3);
}

void f(int i)
{
  int bb[7]={1,2,3,4,5,6,7};
  for (int j=0;j<10;j++)
    {
     printf("%d\n", bb[j]);
     j += 2;
     }
}
```

1.1
2.4
3.7

```
#include <stdio.h>

int A[5][5] = { 2, 3, 5, 7, 11,
    13, 17, 19, 23, 29,
    31, 37, 41, 43, 47,
    53, 59, 61, 67, 71,
    73, 79, 83, 89, 97};

printf("%d\n",A[3][2]);
printf("%d\n",A[2][13]);
printf("%d\n",*( A + 2 ));
printf("%d\n",*( *( A ) + 2 ));
```

1.61
2.89

```
# difference is 256 so they are equivalent
char a = 250;
char b = -6;

if(a == b) printf("%s\n", "Really?");
# outputs "Really?"
# although, if their difference is not 256, they are not equivalent.
```

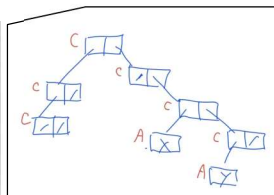
char = (real number) / 256
int = (real number) / 232
long = (real number) / 264

overflow occurs at 2^31

```
char upper[26] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

int main(){
        printf("%c\n", upper['k'-'a']);
        # prints K
        printf("%c\n", *(upper + 'k'-'a'));
        # prints K
        printf("%c\n", *(upper+100));
        # prints something random. UNDEFINED
}
```

```
int main(){
        int *p = new int;
        *p = 3
        cout << *p << endl; // prints 3
        delete p;
        cout << *p << endl; // prints random numbers
        float *q = new float;
        *q = 5; // may overwrite space where p points
        cout << *p << endl; // undefined behavior because p has been reclaimed
```



Your Answer:

(((())) () X Y)

```
printf("%2.1f", matrix[3][4]); // prints 34.0
double* dp = matrix[3]
printf("%2.1f %2.1f\n", *dp, dp[4]); // prints 30.0 34.0

printf("%2.1f %2.1f %2.1f\n", *(dp + 4), *(dp + 14), matrix[3][14]);
// prints 34.0 44.0 44.0

printf("%f\n", **(matrix + 2)); // prints 20.0
printf("%f\n", *( *(matrix + 2) + 3)); // prints 23.0
```

undefined behavior: C doesn't define "expected behavior."

- index out of bounds
- assigning vars randomly

Reference counts

For every data in heap you remember a reference (how many pointer point to that data). If it reaches 0, you know it is garbage.

Mark and sweep

Considers all objects in the heap as potentially reachable. Then everything directly accessible by the stack is marked. Everything unmarked is collected by garbage collector and frees up memory for future allocation.

Stop and copy

Divides the available memory into two semispaces: the "from-space" and the "to-space." The basic idea is to move live objects from the from-space to the to-space, leaving behind only the live objects in the to-space. Reduces fragmentation.

Fragmentation - where memory becomes divided into small, non-contiguous blocks, making it challenging to allocate a contiguous block of memory for a new object or data structure.

Generational

automatic memory management that takes advantage of the observation that most objects have a short lifespan. It divides the heap into multiple generations based on the age of objects, and it applies different garbage collection strategies to each generation. The two main generations are the young generation and the old generation.

runtime error:

- div by 0.
- dereferencing null ptr.
- memory leaks (not deallocating)
- undefined vars.

Side effects: An expression has a side effect if it alters program state dependent on evaluation order, context, & short circuit logic

```
if (3 < 2)< 1) {
  printf("words");
}
printf("different word");
```

3<2 is false so it evaluates to 0

0<1 so its true even though its clearly false prints this

C example

if this was python expressions like 3<2<1 evaluate like normal so it would be false

When expressions have || /&& it can prematurley exit the expression causing parts of the expression to not run

```
if (true || (1/0)) {
    Printf("True");
}
```

expression is true because at least one part is true.

Can't divide by 0 so this WOULD be an error, but it never gets evaluated

most modern languages like C/python have short circuiting, but older languages like Pascal, Ada, Fortran don't

Scheme side effects: avoid by using pure functions

A pure function
- Returns the same output every time for the same input
- No side effects
- has **referential transparency**: it can be replaced with its value without changing program behavior

Impure functions will:
- Modify local/global variables (x++, set!)
- Preform I/O (display, read, printf)
- Call another impure function
- Depend on non-local state (global counter, randomness)

not pure

define global x

```
(define x 0)
(define (impure-add a)
  (set! x (+ x a))
  x) ; impure: modifies global state
```

modify global variable

```
(define (square x)
  (* x x)) ; pure: no state change, same result always
```

A side effect in scheme is any operation that:
- Changes a variable (set!, x++)
- Prints an output (display, newline)
- Reads input or manipulates external state (I/O, files, etc.)

No loops in scheme since no iterator so we have recursion (usually tail)

```
(define (trfac n)
  (letrec ((helper (lambda (result m)
                     (if (= m n) result
                         (helper (* result (+ m 1)) (+ m 1))))))
    (helper 1 0)))
```

number to compute factorial of

accumulation so far

curr number to n

starts recursion

recursive call at the end

This is a **tail-recursive loop** that avoids side effects.

Mutating a counter variable like in C is avoided. (so its pure)

Scheme syntax

1. Function Definition

```
(define (name arg1 arg2 ...) (body))
```

Example:
- Surrounds function
- logic
- (funcName inputs)

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

2. Recursion

Recursive functions call themselves to process lists, compute factorials, etc.

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

define function with input n

evaluates factorial

returns n * n-1

not tail recursion since its doing math after recursive call (* n (...))

3. Tail Recursion

Optimized form of recursion where the recursive call is the **last** operation.

computes factorial with tail recursion

```
(define (trfac n)
  (letrec ((helper (lambda (result m)
                     (if (= m n) result
                         (helper (* result (+ m 1)) (+ m 1))))))
    (helper 1 0)))
```

function definition

create lambda function to define helper func.

helper function

calls helper to start the recursion (still inside trfac)

if m=n return result

else return recursive call

tail recursion since the last thing it does is the call

4. List Operations

Expression	Description
`'()`	Empty list
`(null? l)`	Checks if list is empty
`(cons a b)`	Prepends `a` to list `b`
`(car l)`	First element of list
`(cdr l)`	Rest of list (drops first)
`(append l1 l2)`	Joins two lists

Example:

```
(cons 1 '(2 3)) ; → (1 2 3)
(car '(a b c))  ; → a
(cdr '(a b c))  ; → (b c)
```

7. Arithmetic and Boolean Operators

Operator	Purpose
`+ - * /`	Arithmetic
`= < > <= >=`	Comparison
`and or not`	Logic
`if`	Conditional

8. Lazy Evaluation

Use `delay` to defer computation, and `force` to compute it later.

Example: Infinite list of integers

```
(define lazy-int
  (letrec ((next (lambda (n) (cons n (delay (next (+ n 1)))))))
    (next 1)))
```

returns tuple (num, promise)

helper

force(cdr gen) to get next

essentially delayed func. call

call recursion

To get the first element:

```
(car lazy-int) ; → 1
(force (cdr lazy-int)) ; → rest of stream
```

lazy-int is a tuple (n, promise)

(car (force (cdr lazy-int))) to get next

(car (force (cdr (force (cdr (lazy-int)))))) for next

10. Special Forms Summary

- `define`: define a function or variable
- `lambda`: anonymous function
- `letrec`: for recursive local definitions
- `if`: conditional
- `begin`: evaluate multiple expressions in order
- `delay`, `force`: for lazy evaluation
- `quote`: prevents evaluation.
- `eval`: forces evaluation of quoted expression.

apply: (apply f arg-list) calls f with those arguments so: (apply f '(a b c))

(f a b c)

also removes quote so it is evaluated

```
(define (print-first-n gen n)
  (when (> n 0)
    (display (car gen))
    (newline)
    (print-first-n (force (cdr gen)) (- n 1))))
; Usage:
(print-first-n lazy_int 5)
```

function to print n numbers

like \n

recursive call

gets promise

forces execution

`apply` — Calls a function with a list of arguments (as if they were separate)

`map` — Applies a function to each element of a list

look bottom right for example

Quiz 6

2.

```
(if #f (+ 9 8) (- 8 6)) → 2
```

False

3.

```
(car (cdr '(car (cdr (UT OCU))))) → '(cdr (UT OCU))
```

list

2nd → 2 choices since '( )

since its a string from '( ) its out put as is

4.

```
(+ (* 3 2) 5) → 11
```

6 + 5

5.

```
(car (cdr '(8 9 10 11))) → 9
```

6.

```
(if (null? '(-2)) + - 6 2) → 4
```

is null? False

if true

Quiz 7

1.

```
(eval '(car '(3 5 6))) ; → 3
```

Forces evaluation

2.

```
((if (< 3 4)
     (lambda (x) (+ x 3))
     (lambda (x) (* x 5)))
 6) ; → 9
```

if true

if false

applies to x of selected lambda function

apply examples:

(apply + '(1 2 3)) → (+ 1 2 3) → 6

(apply (lambda (a b) (+ a b)) '(4 5))

(+ 4 5) → 9

3.

```
(map (lambda (x) (* 3 x)) '(2 3 6)) ; → '(6 9 18)
```

map → apply to each element in list

2·3

3·3

6·3

4.

```
(eval (apply (lambda (x) (cdr x)) '((* + 2 0 4)))) ; → 24
```

5.

```
(define (sqr_list lst)
  (map (lambda (x) (* x x)) lst))
```

function to square all elements in list (Q6)

input list

maps function onto list

(sqr_list '(1 2 3 4)) → '(1 4 9 16)

```
(let ((x 5)
      (y x))
  (squaresum (lambda (a b) (+ (* a a) (* b b)))
             (squaresum y y))) ; → 25
```

creates these x & y at the same time so y goes to the only x that exist

← so essentially just focus on how let statements are nested to see how variables should be set.

1. *(infinite lazy list of cubes)*

Quiz 8

```
(define lazy_cube
  (letrec ((next (lambda (n)
                   (cons (* n n n) (delay (next (+ n 1)))))))
    (next 1))) ; → (1 8 27 64 ...)
```

2. *(tail-recursive square of list elements)*

```
(define (TR_sqr_list l)
  (letrec ((helper (lambda (result l)
                     (if (null? l)
                         result
                         (helper (append result (list (* (car l) (car l))))
                                 (cdr l))))))
    (helper '() l))) ; → (3 0 6) → (9 0 36)
```

1. *(lambda indexing with string)*

Quiz 9 → Python

```
print((lambda x, y: x[y])("Arkansas", 3))  # → 'a'
```

2. *(conditional reassignment of list)*

```
y = 7
x = [5, 7]
if y == 7:
    x = [9, 10, 7]
print(x[1])  # → 10
```

3. *(global list mutation inside function)*

```
x = ['Ohio', 2]
def f():
    x[1] = [2, 4]
print(x)  # → ['Ohio', 2]
f()
print(x)  # → ['Ohio', [2, 4]]
```

4. *(list comprehension with filtering, append, and transformation)*

```
l = [x + "T" for x in "colorado" if x != "o"]
print(l)        # → ['cT', 'lT', 'rT', 'dT']
l.append(7)
l = [3 * x for x in l]
print([l[-1]])  # → 21
```

5. *(list comprehension of numbers that are odd squares + 1)*

$((2 \cdot n)+1)^2+1$ where n goes from 1 to 100

```
[(2 * n + 1)**2 + 1 for n in range(100)]  # → [2, 10, 26, 50, ...]
```

1. *(char comparison with potential wraparound)*

Quiz 12

```
char a = 254;
char b = -2;
if (a == b)
    printf("%d\n", 33);
else
    printf("%d\n", 101);  // → 33
```

2. *(assignment inside `if` condition)*

```
int a = 3;
int b = 0;
if (a = b)
    printf("%d\n", 3323);
else
    printf("%d\n", 13);  // → 13
```

3. *(chained comparison logic)*

```
if (-3 < 3 < 2)
    printf("%d\n", 37);
else
    printf("%d\n", 24);  // → 37
```

Lambda in Python

```
lambda arg1, arg2, ... : expression
```

- It defines an **anonymous (unnamed) function**.
- The expression is automatically **returned** — no `return` keyword needed.
- You can call it immediately or assign it to a variable.

```
add = lambda x, y: x + y
print(add(2, 3))  # → 5
```

It's for **quick, simple functions**, often used in:

- `map`, `filter`, `sorted`, etc.

It can't contain:

- Statements (like `print`, `if`, `while`)
- Multiple expressions

Best for **one-liners** where defining a full function would be overkill

```
# input integer n, calculates the
# sum of the first n integer squares. So
# sumOfSquares(1) = 1*1 = 1
# sumOfSquares(2) = 1*1 + 2*2 = 5
# sumOfSquares(3) = 1*1 + 2*2 + 3*3 = 14

def sumOfSquares(n):
    if n == 1:
        return 1
    else:
        return n*n + sumOfSquares(n-1)

# generator sumOfCubes that produces all
# the sums of consecutive integer cubes
# The first two sumOfCubes are
# 1 (= 1*1*1), 9(= 1*1*1+2*2*2)
def sumOfCubes():
    i = 1
    total = 0
    while True:
        total += i * i * i
        yield total
        i += 1
```

```
#include <stdio.h>
void f(int i);

int main()
{
  f(3);
}

void f(int i)
{
  int bb[7]={2,3,4,5,6,7,8};
  for (int j=0;j<10;j++)
    {
     printf("%d\n", bb[j]);
     j += 2;
    }
}
```

Arrays in C are just **contiguous memory blocks**

Accessing `bb[9]` doesn't check bounds — it just reads whatever is 2 integers past the valid array

You could:

- Overwrite important variables
- Leak or corrupt memory
- Trigger **security vulnerabilities**

```
class University:
    def __init__(self, nm, lo, pr):
        self.nm = nm
        self.lo = lo
        self.pr = pr

    def __call__(self, cn, dean):
        return College(self.nm, self.lo, cn, dean)

class College:
    Purpose = "Professional Education"
    def __init__(self, un, lo, cn, dean):
        self.un = un
        self.lo = lo
        self.cn = cn
        self.dean = dean

um = University("U. of Michigan", "Ann Arbor", "John Cruz")
um.Purpose = "Research"
ocu = University("U. of North Texas", "Edmond", "Bob Betz")
ocucoa = ocu("College of Arts", "Christie")
ocucoa.lo = "Austin"

print(um.nm) # -> U. of Michigan
# print(ocu.Purpose) -> error, purpose is not defined for ocu so
    when it is called it will give an error
# purpose is defined for the class college not the class university
print(ocucoa.lo) # -> Austin
print(um.Purpose) # -> Research
print(ocucoa.Purpose) # -> Professional Education
```

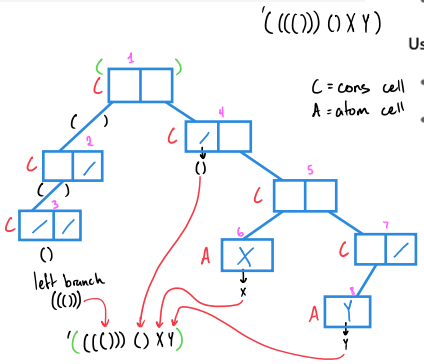
```
for x in range(3):
    x = x+5
    print(x)
print(x)
for x in range(4):
    print(x)
```

1st: 5
2nd: 6
3rd: 7
4th: 7
5th: 0

```c
#include <stdio.h>

int A[5][5] = {
    { 2,  3,  5,  7, 11},
    {13, 17, 19, 23, 29},
    {31, 37, 41, 43, 47},
    {53, 59, 61, 67, 71},
    {73, 79, 83, 89, 97}
};

int main() {
    printf("%d\n", A[3][4]);
    printf("%d\n", A[2][11]);
    printf("%d\n", **(A + 3));
    printf("%d\n", *(*(A) + 4));
    return 0;
}
```



1. Reference Counting

How it works:

Each object has a counter tracking how many references point to it.

- When a reference is created (e.g., `a = obj`), the count increases.
- When a reference is removed (e.g., `a = None`), the count decreases.
- When the count reaches **zero**, the memory is freed.

Pros:

- Simple to implement.
- Works well in real-time systems (no long pauses).
- Memory is reclaimed immediately (predictable cleanup).

Cons:

- **Can't handle cycles** (e.g., two objects referencing each other).
- Requires extra storage for counters and constant updates.

Used in:

- **Python** (main mechanism in CPython), with additional **cycle detection**.
- C++ smart pointers (`shared_ptr`).

2. Mark-and-Sweep

How it works:

1. **Mark phase**: Start at root references and recursively mark all reachable objects.
2. **Sweep phase**: Go through all memory and free anything not marked.

```
text

Heap:
[Root] -> A -> B -> C   (D is unreachable)

Mark phase: A, B, C are marked
Sweep phase: D is collected
```

Pros:

- Handles cyclic references.
- Simple conceptual model.

Cons:

- Can introduce **"stop-the-world" pauses** (program execution halts).
- Not real-time.
- Requires two passes over memory: mark, then sweep.

Used in:

- Java, Scheme, older Lisp implementations.

3. Stop-and-Copy (Copying Collection)

How it works:

- Heap is divided into two equal-sized spaces: **from-space** and **to-space**.
- At any time, allocation happens in one space (from-space).
- When GC occurs:
  - Reachable objects are **copied** to the to-space.
  - The rest (unreachable) are left behind (implicitly freed).
  - The spaces are swapped.

```
text

From-space: [A, B, C (reachable)], [X, Y (unreachable)]
To-space: A, B, C
```

Pros:

- Only traverses reachable objects (efficient).
- Compacts memory (prevents fragmentation).
- No need to sweep all of memory.

Cons:

- Requires twice the memory.
- Objects need to be copied (potential overhead).

Used in:

- Early Lisp, Scheme, educational systems, some JS VMs (like SpiderMonkey).

4. Generational Collection

Key idea:

Most objects die young (e.g., temporary variables).

How it works:

- Divide the heap into multiple **generations** (typically: young, old).
- Objects start in the **young generation**.
- If they survive a collection, they are promoted to the **old generation**.

Collection strategy:

- **Young gen** is collected **frequently** and quickly using a method like stop-and-copy.
- **Old gen** is collected **less often**, usually via mark-and-sweep or other strategies.

Pros:

- Very efficient for most programs (many short-lived objects).
- Reduces pause times by focusing on small, active memory regions.

Cons:

- More complex implementation.
- Needs a "remembered set" for old-to-young references.

Used in:

- Java (HotSpot VM).
- .NET CLR.
- Many modern language runtimes.

5. Pointer Reversal

What it is:

A technique used during depth-first traversal (like in the mark phase of mark-and-sweep) **without using extra memory** (e.g., no recursion stack).

How it works:

- Temporarily overwrite the **next pointer** in an object to point back to its **parent** in the traversal.
- This creates a reversible path through the heap.

```
text

Instead of:
A → B → C

Use:
B → A (temporarily), then restore
```

Pros:

- No need for recursion or explicit stack.
- Saves memory, useful on low-memory systems.

Cons:

- Hard to implement correctly.
- Risk of corrupting the heap if reversed pointers are not restored correctly.
- Only suitable for specific kinds of traversals (e.g., linear data structures).

Used in:

- Some Lisp implementations.
- Theoretical/educational GCs (space-constrained ↓ ironments).

Summary Table

Technique	Handles Cycles	Real-Time Friendly	Needs Extra Memory	Used In
Reference Count	❌	✅ (usually)	Minimal	Python, C++ smart pointers
Mark and Sweep	✅	❌	Moderate	Java, Scheme
Stop and Copy	✅	❌ (but fast)	High (2x heap)	Scheme, JS
Generational	✅	✅ (young gen)	High (multi-gen)	Java, .NET
Pointer Reversal	✅	✅ (space-saving)	❌ (no stack)	Low-memory systems, Lisp