1) Lambda utili
    - ; (lambda(parametri)(corpo)) input ((lambda(x y)(+ x y)) 1 2)
    - ; Con la defparameter, dobbiamo usare "funcall" (defparameter *ciao* (lambda (x y) (+ x y))) (funcall *ciao* 1 2)
    - ; Applicazione mapcar (mapcar (lambda (x)(+ x 1)) '(1 2 3))
    - ; Applicazione removerif (remove-if (lambda (x) (= x 0)) '(1 0 2))
    - ; Find-if (ritorna nil se non trovato) (find-if (lambda(x) (= x 1)) '(1 0 2))
    - ; Position-if: ritorna l'indice, nil se no (position-if (lambda(x)(= x 5)) '(1 2 2))
    - ; Reduce, che riduce 1 lista in 1 funzione (reduce (lambda(x y)(+ x y)) '(1 0 2))
    - ; Every, fa il lambda su tutti gli elementi (every (lambda(x)(>= x 0)) '(1 0 2))
    - ; Subseq (inizio, fine-1) (subseq '(1 2 5 4) 0 1)
    - ; Concatena 2 liste (append '(1 2) '(3 4))
    - ; Ordina 1 lista (sort '(1 2 3 4) #'>)
    - ; Controlla se siamo alla fine di una lista (endp '())
2) Definire una lambda expression e come trasformare una let in una lambda
    - Una lambda expression è una funzione anonima (lambda (parametri) corpo)
    - Serve per creare una funzione senza dover assegnare un simbolo, utili per funzioni temporanee/passare funzioni come argomenti ad altre funzioni
    - Trasformiamo una let in una lambda (let ((x 10) (y 20)) (+x y)) ((lambda (x y) (+x y)) 10 20)
        - Prendiamo tutti i parametri della let e li mettiamo come parametri della lambda
        - Riscriviamo il codice della let dentro il corpo della lambda
        - Diamo come input i valori inizali della let
3) Cosa sono le espressioni valutanti
    - Sono espressioni che si valutano a se stesse, quindi vuol dire che non c'è bisogno di elaborazione aggiuntiva
    - Aka, quando mettiamo come input quel valore nella console, ci ritorna lo stesso valore
    - 42 -> 42
    - Nil -> nil
    - T -> T
4) Concetto di closure in lisp e dare definizione
    - E' una funzione che conserva le variabili locali del contesto in cui è stata definita
    - Questo permette di accedere alle variabili anche quando viene chiamato in contesti diversi dall'originale

```
(defun my-adder (grow-by)
  (let ((sum 0))
    (lambda ()
      (incf sum grow-by))))

(defvar *two-counter* (my-adder 2))
(defvar *two-counter-two* (my-adder 2)) ; a second adder by 2
(defvar *three-counter* (my-adder 3))

(funcall *two-counter*)
=> 2

(funcall *two-counter*)
=> 4

(funcall *three-counter*)
=> 3

(funcall *two-counter-two*)
=> 2
```

5) In lisp è sempre più efficente usare un loop ai posto di una funzione?
    - No, non è sempre più efficente, in particolare quando l'algoritmo è terminale
        - Se la ricorsione è ottimizzata ed l'algoritmo è terminale, il compilatore lo può ottimizzare in un loop
    - I loop sono più efficenti quando l'algoritmo non è terminale
6) E' possibile interpretare lisp/prolog senza garbage collector? Qual'è l'operazione fondamentale che alloca memoria in lisp?
    - Si, è possibile, però la memoria ne risentirebbe
    - L'operatore fondamentale che alloca memoria è la cons, e si utilizza per creare nuove coppie di valori e costruire strutture dati come liste
7) Cos'è l'activation frame, cosa succede dentro prima e dopo un return?
    - L'activation frame è una struttura dati per tenere traccia delle informazioni necessarie per gestire la gestione delle chiamate di funzioni
        - E' un blocco di memoria che si trova sullo staco
    - Prima contiene tutte le informazioni necessarie per l'esecuzione della funzione
        - E prepara tutti i valori di ritorno e li prepara per restituire al chiamante
    - Poi dopo esso viene rimosso dallo stack e il programma continua dalla funzione precedente, salvato nel link dinamico
        - Se c'è un activation frame, viene deallocato
    - Esso contiene:
        - Static link
        - Dynamic link
        - Indirizzo ritorno
        - Variabili per variabili
        - Spazi per registri da salvare prima di chiamare sottofunzioni
8) Cosè una cons-cell
    - E' una struttura dati dove abbiamo una coppia di valori che essi possono essere
        - Atomi
        - Altra cons-cell
    - Per prenere il primo valore, si usa car
    - Per prendere secondo valore, si usa cdr
    - Essa è fondamentale per costruire le liste
9) Cosa si intende per ricorsioni in coda e quali tipi di programmi denota? Fornire esempio dove è applicabile e dove non
    - E' un tipo di ricorsione dove l'ultima operazione è la chiamata alla funzione stessa
    - E' ottima siccome il compilatore può renderla iterativa e quindi ottimizzarla
        - Aka devota programma e che possono venire tradotti dai compilatori in cicli
    - Es. Se l'ultima operazione è una moltiplicazione, essa non è applicabile

```
(defun factorial-non-tail-rec (n)
  (if (<= n 1)
      1
      (* n (factorial-non-tail-rec (- n 1)))))
```

    - Questo vuol dire che, nella ricorsione non in coda bisogna mantenere la chiamata della funzione corrente nello stack per l'elevorazione successiva
10) Quanti activation record vengono fatti in una tail recursive vs non tail recursive
    - In una TR, ne basta 1 che si può riutilizzare
    - Nelle non TR, siccome dobbiamo tenere traccia dei dati siccome poi avremo un altra operazione, ne avremo tanti quanti le nostre funzioni ricorsive
    - Nota che, una TR minimizza l'overhead
11) Perché i linguaggi funzionali puri ammettono il passaggio dei parametri ad una funzione solo per valore e non, ad esempio, per riferimento?
    - Siccome questi linguaggi si vogliono basare sul principio di determinismo: stessi input, stessi output
    - I dati sono immutabili, e quindi non si possono vedere come "scatola nera"
12) Dare rappresentazione cons di (a b (c) (d ((e))))
(cons 'a (cons 'b (cons (cons ' c nil) cons(cons(cons 'd (cons (cons(cons(e nil))))))

13) 3 frasi che permettono di applicare il paradigma funzionale
    1) Read -> leggere ciò che viene presentato in input
        - Definizione
    2) Eval -> Prende la struttura dati prodotta nella lettura e la valuta
        - Applicazione
    3) Print -> Prende il risultato e lo converte in rappresentazione leggibile
        - Ugh... Il mega dice composizione...?
        - Copilot/chat gpt mi danno risposte diverse
        - E contemporaneamente non c'è la risposta negl appunti del prof

Una dimostrazione pratica è il ciclo REPL
    - Read
    - Eval
    - Print

E' combina in un ciclo continuo
14) Cosa fa un interprete Lisp
    - Ci che fa è applicare il ciclo REPL (leggere sopra)
15) Quanti momenti ha lisp?
    - Si ripiega il ciclo REPL
16) Cosa significa che lisp è un linguaggio puro?
    - Vuol dire che, stessi input danno lo stesso output e non hanno effetti collaterali
15) Differenza tra simbolo e atomo
    - Un atomo è un tipo di dato primitivo, un entità indivisibile, aka numeri, stringhe e altri dati primitivi. In generale, è una qualunque entità che non è una lista
    - Un simbolo è una stringa di caratteri che rappresenta un identificatore
        - Identificatore/nome di una variabile o funzione
16) Cosa mi aspetto di trovare dopo una parentesi tonda?
    - Una funzione/operatore
    - Se trovassi il simbolo di una variabile dopo la parentesi tonda, cosa produce l'interprete?
        - Se la variabile è una funzione, tornerà la chiamata della funzione
        - Se non lo è, tornerà errore
    - Posso scrivere a con gli argomenti (a 2 4)
        - Solamente se a è una funzione
17) Cos'è una funzione di ordine superiore
    - E' una funzione che prende un altra funzione come parametri e restituire il risultato della funzione passata per parametro

1. Funzione che prende un'altra funzione come parametro

```
(defun apply-twice (f x)
  (funcall f (funcall f x)))

(apply-twice #'1+ 5) ; Applica la funzione 1+ due volte a 5, restituendo 7
```

2. Funzione che restituisce un'altra funzione

```
(defun make-adder (n)
  (lambda (x) (+ x n)))

(setq add3 (make-adder 3)) ; Restituisce 15, poiché add3 è una funzione che aggiunge 3
```

18) Creare una funzione che applica la composizione di 2 funzioni
(defun compose(f g)
    (funcall f (funcall g))
)
    - Se volessi fare una funzione che ritorna la funzione composizione di 2 funzioni (defun compose(f g)
        (lambda(x)
            (funcall f(funcall g x)))
19) Parla del concetto di transparenza referenziale
    - Stessi input, dati ad una funzione, tornano gli stessi output senza effetti collaterali
20) A cosa serve la funcall
    - Serve per chiamare funzioni passate come parametro
    - Utile per le funzioni di ordine superiori
21) Cos'è una ricorsione semplice, doppia e in coda?
    - Ricorsione semplice = Chiamo se stesso
    - Ricorsione doppia = Chiama se stessa più volte in 1 chiamata ricorsiva
    - Ricorsione in coda = L'ultima operazione fatta dalla funzione è la ricorsione
22) Che sottoinsieme di runtime è presente in lisp (e java/js/prolog/haskell/julia) ma non in c/c++?
    - Garbage collection
23) Differenze e comunianze haskell e julia
    - Sono tutti e due due linguaggi di programmazione funzionali che trattano le funzioni come oggetti
    - La differenza è che
        1) Haskell usa una valutazione lazy (tutte le espressioni vengono valutate quando necessarie, questo permette di lavorare con tante strutture dati)
        2) Haskell è staticamente tipizzato, questo vuol dire che le variabili non cambiano di tipo durante l'esecuzione
        3) Julia adotta una valutazione eager (vuol dire che tutto viene valutato quando viene incontrato)
        4) Julia è dinamicamente tipizzato, cioè il tipo delle variabili può cambiare durante l'esecuzione
24) Caratteristiche fondamentali di haskell che lo rendono diverso da lisp e c
    - Haskell ha una valutazione lazy
    - Haskell è tipizzato staticamente
        - Ed ha anche una inferenza di tipi, che vuol dire che i tipi delle espressioni possono non essere esplicitati
    - Haskell è un linguaggio funzionale puro
25) Cos'è il tipo di intarenza ed a quale paradigma di programmazione si riferisce
    - Vuol dire che i tipi delle espressioni sono dedotti durante la compilazione
    - Questo è associato dai linguaggi di programmazione funzionale
26) Dare una definizione di clausola horn
    - E' un tipo speciale di clausola logica che ammette solo 1 letterale positivo
27) Quali sono le operazioni principali che allocano memoria in lisp?
    - List
        - Lista di elementi
    - Cons
        - Testa-coda
        - Car-cdr

28) Cos'è lo heap e la sua importanza nei linguaggi funzionali logici
    - Lo heap è la regione di memoria utilizzata per l'allocazione dinamica degli oggetti
    - Nei linguaggi funzionali, ogni volta che si crea una nuova struttura dati essa viene allocata nello heap
    - Per la gestione di questi dati, viene implementato il garbage collector che serve per rimuovere i dati dallo heap quando non più necessari (non c'è in tutti i linguaggi, tipo non c'è in c e c++ dove lo sviluppatore deve liberare manualmente la memoria)

29) Quali sono le informazioni minime nell'activation frame in lisp?
    - Valori dei parametri
    - Valori locali
    - Indirizzo di ritorno
    - Puntatore al frame precedente