Test PP Numele ..... Grupa .....

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
a																				
b																				
c																				
d																				

1. Care afirmație despre funcțiile `f1` și `f2` este adevărată în Racket?:

```
(define (f1 a b)
  (if (zero? a)
      b
      (let ((newb (+ 1 b)))
        (f1 (- a 1) newb))))
(define (f2 a b)
  (if (zero? a)
      b
      (let ((newf (f2 (- a 1) b)))
        (+ 1 newf))))
```

   (a) `f1` și `f2` calculează același rezultat, dar `f1` are o recursivitate mai eficientă decât `f2`.
   (b) `f1` și `f2` calculează același rezultat, dar `f2` are o recursivitate mai eficientă decât `f1`.
   (c) `f1` și `f2` au același tip de recursivitate, dar există argumente pe care întorc rezultate diferite.
   (d) `f1` și `f2` calculează același rezultat în majoritatea cazurilor, însă din cauza tipului de recursivitate `f1` poate intra în buclă infinită pentru argumente negative.

2. Cum arată procesul generat de apelul `(par? 10)` în Racket?

```
(define (par? x)
  (or (zero? x)
      (impar? (- x 1))))
(define (impar? x)
  (and (> x 0)
       (par? (- x 1))))
```

   (a) Ca o iterație — nu este necesară depunerea de informație pe stivă.
   (b) Ca o recursivitate pe stivă — se așteaptă rezultatele apelurilor pe numere mai mici pentru a putea fi procesate de `or` și `and`.
   (c) Ca o recursivitate arborescentă — exceptând cazul de bază, pentru fiecare apel este nevoie să combinăm două rezultate (unul al funcției `par?`, unul al funcției `impar?`).
   (d) Ca un calcul elementar — funcțiile nu sunt recursive (niciuna dintre ele nu se autoapelează).

3. Ce rezultat produce codul Racket următor?

```
(define (foo f)
  (foldl f 1 '(1 2)))
(define (bar f)
  (foldr f 1 '(1 2)))
(filter (lambda (f) (equal? (foo f) (bar f)))
        (list + - * list))
```

   (a) `(#<procedure:+> #<procedure:*>)`
   (b) `(#<procedure:+> #<procedure:-> #<procedure:*>)`
   (c) `(#<procedure:+> #<procedure:*> #<procedure:list>)`
   (d) eroare

4. Ce rezultat produce codul Racket următor?

```
(define (f a)
  (lambda (b)
    (lambda (g)
      (g b a))))
(map ((f '()) '()) (list cons append list))
```

   (a) `((()) () (() ()))`
   (b) `(() () (() ()))`
   (c) `((() ()) () (()))`
   (d) eroare

5. În evaluarea apelului Racket

```
(apply map (cons list '((1 2 3) (a b c))))
```

funcția `list` este apelată de:

   (a) 3 ori
   (b) 2 ori
   (c) niciodată
   (d) o singură dată

6. Ce rezultat produce forțarea promisiunii `promise` în Racket?

```
(define promise
  (let ([y 1])
    (delay (+ x y))))
(define x 20)
(define y 2)
(force promise)
```

   (a) `21`
   (b) `22`
   (c) `#<promise>`
   (d) eroare

7. Ce rezultat produce expresia Racket de mai jos?

```
(let* ([a 3]
       [b 2]
       [c 1]
       [f (lambda (a b) (+ a b c))])
  (let ([a 4] [b 5] [c 2])
    (f a b)))
```

   (a) `10`
   (b) `11`
   (c) `5`
   (d) `7`

8. Pentru programul Haskell de mai jos:

```
f x = x + 2
g x = x + f 1 + f 1
h x = x + x + x + x + x
```

de câte ori se va evalua corpul funcției `f` la apelul

```
g (f 1) + h (f 1)?
```

   (a) 4
   (b) 1
   (c) 3
   (d) 8

9. De câte ori se evaluează aplicațiile funcției `(+ 1)` în cadul expresiei Haskell de mai jos?

```
length $ map (+ 1) [1..10]
```

   (a) 0
   (b) 1
   (c) 10
   (d) 20

10. Pentru definițiile Haskell:

```
data MyT = Cons1 Int | Cons2 Float | Cons3 Int Float
f cons arg = zipWith (\c a -> c a) cons arg
```

care dintre următoarele legări **NU** generează o eroare?

```
1. let x = f [Cons1, Cons2]
2. let x = f [Cons1, Cons3 1]
3. let x = f [Cons2, Cons3 2]
```

   (a) Doar 3.
   (b) Toate generează eroare.
   (c) Doar 2.
   (d) Doar 1.

11. Care este tipul următoarei expresii în Haskell?

```
[map, filter]
```

(a) `[(Bool -> Bool) -> [Bool] -> [Bool]]`
(b) `[((a -> b) -> [a] -> [b]), ((a -> Bool) -> [a] -> [a])]`
(c) `[(a -> b) -> [a] -> [b]]`
(d) `[(a -> Bool) -> [a] -> [a]]`

12. Funcția `fmap` din Haskell are tipul:

```
Functor f => (a -> b) -> f a -> f b.
```

Acest lucru înseamnă că:

(a) `fmap` este polimorfică ad-hoc în variabila de tip `f` și polimorfică parametric în `a` și `b`.
(b) `fmap` este polimorfică parametric în variabila de tip `f` și polimorfică ad-hoc în `a` și `b`.
(c) `fmap` este polimorfică parametric exclusiv.
(d) `fmap` este polimorfică ad-hoc exclusiv.

13. Care variantă de definire a funcției `null` din Haskell este mai potrivită și de ce?

```
null1 [] = True
null1 _  = False

null2 l = l == []
```

(a) `null1`, pentru că `null2` are un tip mai restrictiv.
(b) `null1`, pentru că `null2` nu acoperă toate cazurile
(c) `null2`, pentru că `null1` nu acoperă toate cazurile.
(d) Ambele variante au exact aceeași funcționalitate și exact același tip.

14. Ce va afișa Prolog dacă îi solicităm satisfacerea scopului `pred(X)` în toate modurile posibile?

```
p(1). p(2). p(3).
q(1). q(3).
pred(X) :- p(A), q(A), p(B), q(B), append(X, Y, [1,2,3,1]),
           member(A, X), member(B, Y).
```

(a) `X = [1] ; X = [1, 2] ; X = [1, 2, 3] ; X = [1] ; X = [1, 2] ; X = [1, 2, 3] ; false.`
(b) `X = [1] ; X = [1, 2] ; X = [1, 2, 3] ; false.`
(c) `X = [1] ; X = [1, 2] ; false.`
(d) `X = [ 1 | _ ] ; Y = [ _ | 1 ] ; false.`

15. Pentru un graf orientat descris în Prolog prin fapte de tip `arc/2` (unde primul argument este nodul sursă și al doilea este nodul destinație), care este o metodă corectă de a genera (nu doar verifica) toate nodurile care au minim doi succesori?

```
manySucc1(X) :- arc(X,Y), !, arc(X,Z), Y \= Z.
manySucc2(X) :- arc(X,Y), arc(X,Z), Y \= Z, !.
```

(a) Niciuna.
(b) Doar `manySucc1`.
(c) Doar `manySucc2`.
(d) Ambele.

16. Ce va afișa Prolog pentru interogarea de mai jos?

```
?- A=[1,2,3], B=[2,3,4],
   findall(X, (member(X, A), member(X, B), !), L).
```

(a) `A = [1, 2, 3], B = [2, 3, 4], L = [2].`
(b) `A = [1, 2, 3], B = [2, 3, 4], L = [2,3].`
(c) `A = [1, 2, 3], B = [2, 3, 4], L = [3].`
(d) `false.`

17. Pornind de la programul Prolog

```
p(1). p(2). p(3).
q(X) :- !, X >= 2.
q(_).
t(X) :- p(X), q(X).
```

ce produce interogarea `t(1)`?

(a) `false`, pentru că `1` nu este mai mare sau egal cu `2` și `q` nu mai caută alte soluții.
(b) `true`, pentru că predicatul `q` este adevărat pentru orice.
(c) `false`, pentru că am folosit *underscore* în a doua regulă a predicatului `q`.
(d) `true`, pentru că este suficient ca unul dintre `p(1)` și `q(1)` să fie adevărate.

18. Care variantă de *sau* se oprește la primul *true* (ceilalți operanzi ai lui *sau* nu au niciun efect dacă primul operand e adevărat)?

(a) `or` din Racket și `||` din Haskell.
(b) `or` din Racket, `||` din Haskell și `;` din Prolog.
(c) Doar `||` din Haskell.
(d) `or` din Racket și `;` din Prolog.

19. Care limbaje permit aplicarea unei entități asupra ei înseși?

```
Racket: (define (f x) (x x))
Haskell: f x = x x
Prolog: x(x).
```

(a) Racket și Prolog.
(b) Racket și Haskell.
(c) Haskell și Prolog.
(d) Niciunul.

20. Care limbaje permit existența elementelor cu tip diferit în cadrul aceleiași liste?

(a) Racket și Prolog.
(b) Racket și Haskell.
(c) Haskell și Prolog.
(d) Toate.