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

A	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 dintre următoarele trei programe intră în buclă infinită?
Racket:

```
(define (loop x) (loop x))
(define (p x)
  (if (zero? x) (loop x) 1))
```

Haskell:

```
loop x = loop x
p 0 = loop 0
p _ = 1
```

Prolog:

```
loop(R) :- loop(R).
p(0, R) :- loop(0).
p(N, 1) :- N > 0.
```

   (a) Niciuna dintre celelalte variante
   (b) Programul Haskell, prin evaluarea expresiei `p 2`
   (c) Programul Racket, prin evaluarea espresiei `(p 2)`
   (d) Programul Prolog, prin interogarea `p(2, R).`

2. Pentru funcția Racket `f` de mai jos, la ce se evaluează expresia `(f * even? '(1 2 3 4))`?

```
(define (f g h L)
  (apply +
         (apply g (filter h L))
         (apply g (filter (compose not h) L))
         '(5 6)))
```

   (a) `21`
   (b) `(8 13)`
   (c) `22`
   (d) `(8 1 5)`

3. Pentru predicatul `del` în Prolog, câte soluții produce interogarea `del(X, A, [1,2]), del(X, B, A).`?

```
del(X, [X|T], T).
del(X, [Y|T], [Y|R]) :- del(X, T, R).
```

   (a) 2
   (b) 6
   (c) 9
   (d) 12

4. Care dintre următoarele definiții generează în Racket fluxul în care fiecare element este perechea `(2 . 3)`?

```
(define stream-1 (stream-cons (cons 2 3) stream-1))
(define stream-2 (cons (cons 2 3) stream-2))
(define stream-3
  (let ([p (cons 2 3)])
    (stream-cons p
      (stream-map (lambda (x) (cons (car x) (cdr x)))
                  stream-3))))
(define stream-4 (stream-cons (list 2 3) stream-4))
```

   (a) `stream-3` și `stream-4`
   (b) `stream-1` și `stream-3`
   (c) `stream-1` și `stream-4`
   (d) Doar `stream-1`

5. La ce se evaluează expresia Haskell de mai jos?

```
foldl (\acc x -> [x]:acc) [] [1,2,3]
```

   (a) `[1,2,3]`
   (b) `[[1],[2],[3]]`
   (c) `[[3],[2],[1]]`
   (d) `[3,2,1]`

6. Ce rezultat se obține în Prolog pentru interogările:

```
?- length(L1, 2), length(L2, 3), append(L1, L2, L1).
?- append(L1, L2, L1), length(L1, 2), length(L2, 3).
```

   (a) `false.` în ambele cazuri
   (b) Buclă infinită, respectiv `false.`
   (c) Buclă infinită în ambele cazuri
   (d) `false.`, respectiv buclă infinită

7. Fie următoarea definiție a tipului unui punct în Haskell:

```
data Point
    = Cartesian { xcoord :: Double, ycoord :: Double }
    | Polar     { radius :: Double, theta  :: Double }
```

Cum poate fi rezolvată eroarea de tip a expresiei `Cartesian 3.0 . radius $ Polar 1.0`?

   (a) Ambii operatori ar trebui să fie `$`
   (b) Ambii operatori ar trebui să fie `.`
   (c) Operatorii `.` și `$` ar trebui interschimbați
   (d) Expresia are un tip valid

8. Ce produc următoarele două expresii în Racket,

```
(letrec ([x (map + y y)]
         [y '(1 2 3 4 5)])
  x))
```

respectiv Haskell?

```
let x = zipWith (+) y y
    y = [1..5]
in x
```

   (a) `[2, 4, 6, 8, 10]`, respectiv eroare
   (b) `[2, 4, 6, 8, 10]` în ambele cazuri
   (c) Eroare în ambele cazuri
   (d) Eroare, respectiv `[2, 4, 6, 8, 10]`

9. Care două expresii de tip **NU** unifică în Haskell?

   (a) `(String, a)` cu `([a], Char)`
   (b) `(a, b, [b])` cu `(a -> b, Int, [Int])`
   (c) `(a -> a)` cu `(b -> a)`
   (d) `(a -> Maybe a)` cu `(Int -> t a)`

10. Ce rezultat se obține în Prolog pentru interogarea `findall(R, x(1,R), L1), findall(R, x(3,R), L2).`?

```
y(1). y(2). y(1+2).
z(2). z(3). z(3-2).
x(X, R) :- y(X), !, z(Z), R = Z.
```

   (a) `L1 = [2, 3, 1], L2 = [].`
   (b) `L1 = [2, 3, 1], L2 = [2, 3, 1].`
   (c) `false.`
   (d) `L1 = [2, 3, 3-2], L2 = [].`

11. Știind că funcțiile Haskell `f1` și `f2` de mai jos se aplică întotdeauna pe o listă de întregi care conține cel puțin un număr impar, alegeți afirmația adevărată:

```
keepOdd v
    | odd v     = Just v
    | otherwise = Nothing
f1 xs = let Just v = head (map keepOdd xs) in v
f2 (x:xs)
    | Just v <- keepOdd x = v
    | otherwise           = f2 xs
```

   (a) Doar `f1` extrage primul număr impar din listă pentru orice intrare corectă.

(b) `f1` și `f2` extrag primul număr impar din listă pentru orice intrare corectă.
(c) Doar `f2` extrage primul număr impar din listă pentru orice intrare corectă.
(d) Nici `f1`, nici `f2` nu extrage primul număr impar din listă pentru orice intrare corectă.

12. Fie următoarele definiții în Haskell:

```
xs = (map . map) (+ 1) [[1..5],[6..10]]
ys = map length xs
```

După evaluarea expresiilor `head ys` la 5 și `length ys` la 2, care variantă reflectă starea curentă în evaluarea lui `xs` (nu `ys`)?

(a) `[_, [_,_,_,_,_]]`
(b) `[[_,_,_,_,_],_]`
(c) `[[_,_,_,_,_],[_,_,_,_,_]]`
(d) `[_,_]`

13. De câte ori se evaluează în Haskell corpul funcției `f` la evaluarea `test1`, respectiv `test2`?

```
f x = x * x
lst1 = repeat $ f 2
lst2 = map f $ repeat 2
test1 = take 2 $ drop 3 lst1
test2 = take 2 $ drop 3 lst2
```

(a) De două ori la `test1`, de două ori la `test2`.
(b) O dată la `test1`, o dată la `test2`.
(c) De două ori la `test1`, o dată la `test2`.
(d) O dată la `test1`, de două ori la `test2`.

14. Ce puteți afirma despre codul Racket următor?

```
(letrec ([even? (lambda (n)
                  (or (= n 0) (odd? (- n 1))))]
         [odd?  (lambda (n)
                  (and (not (= n 0)) (even? (- n 1))))])
  (even? 4))
```

(a) Construcția `letrec` depune contextul lui `odd?` și `even?` în contextul global, eliminând necesitatea de a reține acest context pe stivă.
(b) Racket poate aplica *tail-call optimization*, întrucât în urma primului test efectuat de `or`/`and`, aplicațiile lui `odd?` și `even?` ajung practic în poziție finală.
(c) Racket nu poate aplica *tail-call optimization*, întrucât aplicațiile lui `odd?` și `even?` nu sunt în poziție finală.
(d) Nu se intră în recursivitate, întrucât aplicația `(even? 4)` referă funcția `even?` de bibliotecă, care nu este recursivă.

15. Fie un graf orientat aciclic în Prolog dat prin fapte de tipul `nod(X)` și `arc(X/Y)`. Verificarea că în graf **NU** există o cale de cel puțin două muchii este:

(a) Niciuna dintre variante
(b) `forall(nod(X), forall(arc(X/Y), \+ arc(Y/_)))`
(c) `forall(nod(X), forall((arc(X/Y), !), \+ arc(Y/_)))`
(d) `forall(nod(X), forall(arc(X/Y), \+ arc(X/_)))`

16. Presupunem că definim în Haskell o clasă numită `DoubleFoldable`, cu funcțiile `doubleFoldl` și `doubleFoldr`, care își propune să mimeze comportamentul clasei `Foldable`, însă cu doi acumulatori în loc de unul. Ce tip ar avea `doubleFoldr`? Se știe că
`foldr :: Foldable f => (a -> b -> b) -> b -> f a -> b`.

(a) `Foldable f => (a -> b -> c -> b -> c) -> b -> c -> f a -> b -> c`
(b) `DoubleFoldable f => (a -> b -> c -> b) -> b -> f a -> b`
(c) `DoubleFoldable f => (a -> b -> c -> b -> c) -> b -> c -> f a -> b -> c`
(d) `DoubleFoldable f => (a -> b -> c -> (b, c)) -> b -> c -> f a -> (b, c)`

17. Ce puteți afirma despre clasa de mai jos în Haskell?

```
class MyClass c where
    f :: c -> c a
```

(a) Poate fi instanțiată cu constructorul de tip `Maybe`.
(b) Poate fi instanțiată cu tipul `Maybe a`.
(c) Este definită incorect, întrucât clasele nu pot fi parametrizate cu constructori de tip.
(d) Este definită incorect, întrucât variabila `c` apare simultan aplicată și neaplicată în tipul lui `f`.

18. Alegeți afirmația corectă despre funcția Racket de mai jos:

```
(define (f x)
  (lambda (x)
    (lambda (x)
      (+ x x x))))
```

(a) `f` este o funcție *curried* validă al cărei echivalent *uncurried* este `(define (f x y z) (+ x y z))`.
(b) `f` este o funcție *curried* validă al cărei echivalent *uncurried* este `(define (f x y z) (+ z z z))`.
(c) `f` este o funcție *curried* validă al cărei echivalent *uncurried* este `(define (f x x x) (+ x x x))`.
(d) `f` nu este o funcție *curried* validă.

19. Cum ar trebui să fie lungimea listei `L` astfel încât expresiile Racket `(foldl - acc L)` și `(foldr - acc L)` să producă același rezultat indiferent de valorile numerice ale elementelor? Funcția binară primită ca parametru de ambele funcționale ia parametrii în ordinea element-acumulator.

(a) 0 sau impară
(b) Doar pară
(c) Doar impară
(d) Doar 0

20. Alegeti cea mai precisă afirmație despre următoarea funcție Racket:

```
(define (f L acc)
  (cond
    [(null? L) acc]
    [(null? (cdr L)) (+ (car L) acc)]
    [else (f (cdr L) (+ (f (cddr L) acc)))]))
```

(a) Funcția utilizează recursivitate arborescentă pentru că este aplicată de două ori.
(b) Funcția utilizează recursivitate pe coadă întrucât folosește un acumulator.
(c) Funcția utilizează recursivitate pe coadă întrucât nu se mai realizează alte operații dupa aplicația recursivă a funcției.
(d) Funcția utilizează recursivitate pe stivă, liniară.