O funcție nu poate returna într-un caz o valoare de un tip și alteori o valoare de alt tip.

Limbajele C și ML sunt *static tipizate*: tipul valorii asociate unui identificator (nume) poate fi determinat static, la compilare, înainte de a rula programul. În C, tipurile trebuie date explicit (pentru variabile, parametri, rezultate de funcție). În ML, ele pot fi deduse de compilator.

Tipizarea statică ne obligă ca valoarea returnată de o funcție să fie în toate cazurile de același tip. În C, tipul e declarat înaintea numelui funcției. În ML, de exemplu, dacă am scris pe o ramură `function | [] -> 0` înseamnă că funcția trebuie să returneze *întotdeauna* un întreg, și nu putem scrie pe altă ramură `| h :: t -> h :: f (n-1) t`, pentru că aici se returnează o *listă*.

Valoarea returnată de o funcție trebuie să fie de tipul cerut în problemă.

Problemele de examen cereau să se returneze: o valoare dintr-o listă, un număr de elemente dintr-o listă, o listă, o pereche de liste, sau doar să tipărească. Toate acestea sunt tipuri diferite. Unii din voi au scris pentru trei probleme diferite același tipar: `function | [] -> []`, sau `function | [] -> 0`. După cum am discutat înainte, asta determină tipul funcției *în toate cazurile*. O funcție începută cu primul tipar va returna *întotdeauna o listă*, deci e incorectă pentru o problemă care nu cere liste. *Numărul* de elemente și un *element* nu pot fi comparate pentru că lista s-ar putea să nu fie de numere.

Valoarea `()` înseamnă "nimic" și e unica valoare de tipul `unit`. Este valoarea returnată de toate funcțiile de tipărire (folosite pentru *efectul* lor, nu pentru valoarea returnată). O funcție cu tiparul `function | [] -> ()` trebuie să returneze `()` în toate cazurile, deci nu va avea vreo valoare utilă.

O excepție și un mesaj de eroare nu sunt același lucru.

Unele funcții nu au valori definite pentru toate argumentele. Funcția `List.hd` care dă primul element al unei liste nu poate returna o valoare pentru lista vidă. În aceste cazuri, funcția generează o *excepție* (în cazul dat, `Failure "hd"`, unde `Failure` e o excepție predefinită, cu un argument șir). Excepția încheie execuția funcției, *fără a returna un rezultat*, și poate fi tratată în fragmentul de program care a apelat funcția. A returna sau a tipări un mesaj de eroare *nu este* același lucru, ci reprezintă în continuare un rezultat, care determină tipul funcției (în toate cazurile). Dacă scriem `function | [] -> "eroare lista vida"`, funcția e obligată să returneze tot timpul un șir. Scriind `function | [] -> print_string "eroare lista vida"`, funcția poate returna doar `()` (rezultatul tipăririi). În ambele cazuri, funcția nu mai poate returna pe cealaltă ramură primul element al listei. Pentru aceasta ne trebuie *excepțiile*, care se pot folosi indiferent de tipul funcției. Scriem:
`function | [] -> raise (Failure "lista vida")`, sau folosim funcția `faiwith` care generează aceeași excepție: `function | [] -> failwith "lista vida"`.

În limbajele funcționale pure nu avem atribuiri.

Unii din voi au scris cod de genul `... -> cnt = cnt + 1; preorder cnt ...`, încercând să modifice `cnt` prin atribuire. În limbajele funcționale pure nu există și nu avem nevoie de atribuire. Ceea ce vrem e să folosim mai departe o nouă valoare. Pentru aceasta, e suficient să dăm noua valoare ca parametru: `... preorder (cnt + 1) ...` Dacă valoarea respectivă va fi folosită de mai multe ori, putem să îi dăm un *nou* nume: `let cnt1 = cnt + 1 in printf "%d" cnt1; preorder cnt1 ...`

O relație nu e doar o pereche.

O relație binară între A și B e o *submulțime* $R \subseteq A \times B$ a produsului cartezian $A \times B$. Deci, R e o *mulțime* de perechi. (a, b) e o pereche, nu o relație, după cum 5 e un întreg, nu o mulțime de întregi. (a, b) poate fi un *element* dintr-o relație: $(a, b) \in R$, uneori notat $R(a, b)$, și chiar *singurul* element din acea relație: $R = \{(a, b)\}$. Dar nu putem scrie $R = (a, b)$. R e o mulțime, (a, b) e o pereche.

Reflexivitatea, simetria, tranzitivitatea sunt definite pentru relații *binare* pe *o* mulțime ($A = B$). (c, b, a) e doar *tripletul* (a, b, c) scris în ordine inversă. Nu are nicio legătură cu o relație binară (și deci nici cu una simetrică, chiar pe o mulțime de 3 elemente), pentru că relația binară conține *perechi*.

Există mulțimi infinite numărabile și nenumărabile.

Mulțimea numerelor naturale e infinită și numărabilă (e chiar punctul de referință pentru a defini mulțimi numărabile). Mulțimea numerelor reale e infinită și nenumărabilă.

Afirmații de genul: "Mulțimea e infinită, deci nenumărabilă", sau "Orice mulțime este finită $\Rightarrow$ este și numărabilă" pun grave semne de întrebare privind abilitatea de a raționa logic.

O afirmație nu se demonstrează doar enunțând definiția.

Considerați afirmațiile (din lucrările de examen):
a) Mulțimea vectorilor de întregi *e numărabilă* pentru că are cardinalul egal cu cardinalul unei submulțimi a numerelor naturale.
b) Mulțimea vectorilor de întregi *e nenumărabilă* pentru că *nu* are cardinalul egal cu cardinalul unei submulțimi a numerelor naturale.

"Are cardinalul egal cu ..." și "nu are cardinalul egal cu ..." sunt două afirmații (contradictorii). Ele sunt făcute luând definiția unei mulțimi numărabile și afirmând *fără niciun argument* că în acest caz condiția din definiție e îndeplinită (respectiv nu). Niciuna din afirmații nu e *demonstrată*. Nu avem nicio bază care să susțină vreuna din ele.

Pentru un alt exemplu, putem afirma pentru numărul
n=1522605027922533360535618378132637429718068114961380688657908494580122963258952897654000350692006139 :
(a) n e compus, fiindcă e produs de doi factori >1; (b) n e prim, fiindcă nu e produs de doi factori >1
Vă convinge vreuna din afirmații? Puteți spune care din ele e adevărată ?
Ambele afirmații încearcă să aplice o definiție (pentru număr compus, resp. prim). Nicicare afirmație nu demonstrează că e *adevărată condiția* din definiție. Pentru (a) ar trebui scris efectiv n ca produs a două numere. Pentru (b) ar trebui verificat că niciun număr între 1 și n nu divide pe n. În fapt, avem
n=37975227936943673922808872755445627854565536638199 × 40094690950920881030683735292761468389214899724061.

În problema din examen, mulțimea vectorilor de numere întregi e într-adevăr numărabilă. Unii au argumentat că numărăm întâi vectorii de lungime 1, apoi cei de lungime 2, etc. Intuiția e parțial bună, dar deja vectorii de lungime 1 sunt în număr infinit (corespund la mulțimea numerelor întregi), deci nu putem să-i enumerăm pe toți întâi. Putem găsi însă o altă ordine de numărare bazată pe aceeași intuiție de a număra întâi vectorii "mici": pentru un vector $v = (a_1, a_2, \ldots, a_k)$ definim o măsură care ține cont și de lungimea vectorului, și de mărimea elementelor: $m(v) = k + |a_1| + \ldots + |a_k|$. Există un număr finit de vectori de măsură n: lungimea și valoarea absolută a fiecărui element sunt cel mult n. Deci putem enumera toți vectorii de întregi în ordine crescătoare a măsurii.

O altă variantă ar fi să demonstrăm întâi că vectorii de k întregi sunt numărabili, ceea ce se face ușor prin inducție, asimilând un vector de lungime $k+1$ cu o pereche dintr-un vector de lungime k și încă un număr, și aplicând același argument ca la numerele raționale. Apoi avem o înșiruire numărabilă (după lungimea k a vectorului) de mulțimi numărabile, care e numărabilă (folosind aceeași construcție diagonală ca pentru numerele raționale).

Echivalența e o implicație în ambele sensuri.

A: Oricine a picat un examen a trecut un examen. B: Nimeni nu a picat toate examenele.

Mulți au afirmat: "Cele două afirmații sunt echivalente pentru că dacă cineva a picat un examen atunci a și trecut unul deci nu a picat toate examenele.". Asta arată (informal) doar că $A \to B$. Pentru ca afirmațiile să fie echivalente, trebuie arătat și $B \to A$.

Două afirmații NU sunt echivalente pentru că implică același lucru.

Unii au afirmat "Propozițiile sunt echivalente deoarece ambele implică aceeași concluzie". Acesta e un raționament fals! Cu această "regulă" de deducție am putea demonstra că două afirmații arbitrare sunt echivalente, deoarece fiecare din ele implică *true* ! Ca un alt exemplu, fie afirmațiile $x = 2$ și $x = 3$. Ambele implică faptul că x e pozitiv. Evident, ele nu sunt echivalente, dimpotrivă, se contrazic!

O definiție trebuie să fie precisă.

O definiție trebuie să fie neambiguă, trebuie să fie clar și precis ce înseamnă, ce corespunde definiției și ce nu. Pentru aceasta, definiția trebuie exprimată riguros, matematic, folosind noțiunile de bază învățate: mulțimi, funcții, relații, formule logice, etc. Nu putem spune doar "o interpretare e încercarea de a da un înțeles unei formule". Nici "un automat e ceva care trece din stare în stare". Nu e suficient nici măcar să spunem "are o funcție de tranziție", e esențial să definim *care e forma* acelei funcții: dacă $\delta : S \times \Sigma \to S$, automatul e determinist (pentru orice pereche de stare și simbol de intrare, starea următoare e unic determinată de funcția de tranziție); dacă $\delta : S \times \Sigma \to \mathcal{P}(S)$, e nedeterminist (pentru orice stare și intrare, funcția permite o mulțime de stări următoare); chiar și mașina Turing are o funcție de tranziție și "trece din stare în stare", dar nu e un automat finit.

Lucrând mecanic, riscați să vă îndepărtați de soluție.

Forma normală conjunctivă are afară $\wedge$ și înăuntru $\vee$. Aplicând distributivitatea $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ însă transformă o formulă care *e* în forma dorită într-o formulă care nu mai e!

Deci, dacă avem $b \to (a \wedge (c \to d)) = \neg b \vee (a \wedge (\neg c \vee d))$ vrem să distribuim pe $\vee$, nu pe $\wedge$: $(\neg b \vee a) \wedge (\neg b \vee \neg c \vee d)$, și am terminat!

Altfel, ne complicăm (și riscul de a greși crește): $\neg b \vee (a \wedge (\neg c \vee d) = \neg b \vee ((a \wedge \neg c) \vee (a \wedge d))$
$= ((\neg b \vee a) \wedge (\neg b \vee c)) \vee (a \wedge d) = (((\neg b \vee a) \wedge (\neg b \vee c)) \vee a) \wedge (((\neg b \vee a) \wedge (\neg b \vee c)) \vee d)$
$= (\neg b \vee a \vee a) \wedge (\neg b \vee c \vee a) \wedge (\neg b \vee a \vee d) \wedge (\neg b \vee c \vee d)$
Am obținut o formă normală conjunctivă, dar e mult mai complicată. Uitându-ne cu atenție, vedem că prima clauză $\neg b \vee a$ e inclusă în a doua, $\neg b \vee c \vee a$ și în a treia, $\neg b \vee a \vee d$, deci prin absorbție ($p \wedge (p \vee q) = p$) putem elimina clauzele 2 și 3, ajungând, mult mai greu, la același rezultat ca înainte.

Gramaticile generează doar șiruri de *terminale*.

O problemă cerea să se scrie șirurile de lungime ≤ 3 generate de gramatica $S ::= AS \mid c$, $A ::= a \mid bA$.

O gramatică definește un limbaj format din șiruri de *terminale* (care se obțin din simbolul de start aplicând regulile gramaticii). Deci un răspuns care enumerează ASc, abA, etc. e din start greșit.

O soluție e să înlocuim pas cu pas fiecare neterminal după fiecare regulă (2x2 opțiuni pentru AS):

$S ::= AS \mid c \;=\; aAS \mid ac \mid bAAS \mid bAc \mid c \;= \ldots$, să eliminăm șirurile de lungime > 3 ($bAAS$, din care fiecare neterminal va genera cel puțin o literă), iar pentru șirurile neterminal rămase aAS și bAc să remarcăm că trebuie să ne limităm la producțiile care dau o singură literă, obținând aac și bac.

O soluție mai sistematică e să tratăm fiecare simbol în parte: S generează AS, apoi AAS, $AAAS$, etc. deci A^nS cu $n \geq 1$. Singura producție care se oprește e $S ::= c$, deci obținem $S = A^nc$, cu $n \geq 0$. Similar, A generează bA, bbA, $bbbA$, ..., deci b^nA cu $n \geq 1$. Combinând cu varianta de oprire $A ::= a$ obținem $A = b^*a$ (un limbaj regulat). Cum A are lungime cel puțin 1, în $S = A^nc$ trebuie ca $n \leq 2$, și obținem: c, ac, bac, aac. Întreg limbajul e de fapt regulat: $(b^*a)^*c$.