

# Programare<sup>de</sup> facultat<sup>e</sup><sub>iva</sub>

Marius Minea

3 martie 2010

# Ce discutăm?

*Probleme: fundamentale, relevante, interesante* (sper)  
azi: SAT checking (valoarea de adevăr a unei formule)

## *Aplicații*

azi: probleme de planificare

*Concepte și tehnici* în proiectarea de programe

azi: recursivitate, programare funcțională, iteratori, interfețe, excepții

## *În rezumat*

**Nu e un curs de:** programare, limbaje, algoritmi (aveți deja)

Caut subiecte complementare, relevante  
(e mai greu decât să ții un curs...)

Puteți *propune voi* subiecte

# Problema teoretică

Se dă o formulă în *logică propozițională*.

Există vreo atribuire de valori de adevăr care o face adevărată ?

= e *realizabilă* (engl. *satisfiable*) formula ?

$$\begin{aligned} & (a \vee \neg b \vee \neg d) \\ & \wedge (\neg a \vee \neg b) \\ & \wedge (\neg a \vee c \vee \neg d) \\ & \wedge (\neg a \vee b \vee c) \end{aligned}$$

Găsiți o atribuire care satisface formula?

Formula e în *formă normală conjunctivă* (conjunctive normal form)

= conjuncție de disjuncții de *literale* (pozitive sau negate)

Fiecare conjunct (linie de mai sus) se numește *clauză*

# De ce e importantă?

## Practic:

În *probleme de decizie* / constrângere:

Putem găsi o soluție la ... cu proprietatea ... ?

⇒ condițiile se pot exprima ca astfel de formule

- ▶ în verificarea de circuite (ex. optimizăm funcția  $f$  în  $f_{opt}$ )  
 $f(v_1, \dots, v_n) = f_{opt}(v_1, \dots, v_n)$  e echivalent cu  
 $f(v_1, \dots, v_n) \oplus f_{opt}(v_1, \dots, v_n) = 0$   
⇒ am optimizat corect dacă  $f \oplus f_{opt}$  NU e realizabilă
- ▶ în verificarea de software (model checking), testare, depanare
- ▶ în biologie (determinări genetice), etc.

# De ce e importantă?

## Teoretic:

E prima problemă demonstrată a fi *NP-completă*.  
(probleme care se crede că nu au soluții polinomiale)

O problemă e *NP-completă* dacă  
o soluție poate fi *verificată* în timp polinomial (e în *NP*)  
(a *verifica* o soluție e mult mai ușor decât a o găsi!)  
dacă se *rezolvă* polinomial, atunci și orice altă problemă din NP.

Cum demonstrăm că o problemă e NP-completă (grea) ?  
reducem o problemă cunoscută la problema studiată  
⇒ dacă s-ar putea rezolva polinomial problema nouă,  
atunci s-ar putea rezolva și problema cunoscută

# Aplicație: Planificarea

= un termen general pentru probleme de luare de decizie

Exemple:

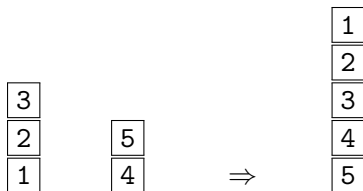
- deplasările unor roboți inteligenți

- comportamentul sistemelor autonome (sonde spațiale)

- rezolvarea de probleme (de tip puzzle, jocuri, etc.)

În general: într-un sistem descris prin *stări* și *acțiuni* (tranziții), cum găsim o cale de la o *stare inițială* la o *stare țintă* (finală) ?

## Exemplu: lumea blocurilor



Acțiuni: mutarea unui bloc liber pe alt bloc.

Ce acțiuni trebuie efectuate ? Care e numărul minim de acțiuni ?

! *Stările* și *tranzițiile* sistemului se pot reprezenta ca *formule logice*

## Reprezentarea unei stări

Putem folosi *propoziții* (variabile boolene):

2

1

3

$p_{2on1} \wedge p_{1on0} \wedge p_{3on0}$  (2 e pe 1; 1 și 3 pe masa)

scriem și propozițiile neadevărate (din totalul de  $n^2$  propoziții)

$$\neg p_{1on2} \wedge \neg p_{1on3} \wedge \neg p_{2on0} \wedge \neg p_{2on3} \wedge \neg p_{3on1} \wedge \neg p_{3on2}$$

Sau: reprezentăm *pe ce* se află fiecare piesă:

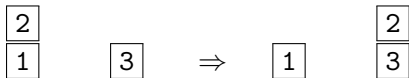
$$base_1 = 0 \wedge base_2 = 1 \wedge base_3 = 0$$

întregi, codificați în binar  $\Rightarrow$  total  $n \log n$  biți (propoziții)

Codificarea mai compactă nu duce neapărat la rezolvare eficientă.



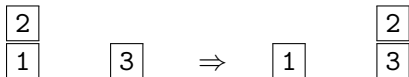
## Reprezentarea unei tranziții



*Efectul* mutării:  $p'_{2on3}$  (notăm cu ' starea următoare)

Constrângeri de execuție (starea anterioară):

## Reprezentarea unei tranziții

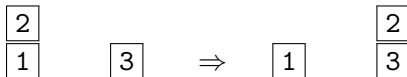


*Efectul* mutării:  $p'_{2on3}$  (notăm cu ' starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$  (piesa mutată e liberă)

## Reprezentarea unei tranziții



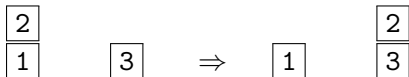
*Efectul* mutării:  $p'_{2on3}$  (notăm cu ' starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$  (piesa mutată e liberă)

$\neg p_{1on3} \wedge \neg p_{2on3}$  (piesa țintă e liberă)

## Reprezentarea unei tranziții



*Efectul* mutării:  $p'_{2on3}$  (notăm cu ' starea următoare)

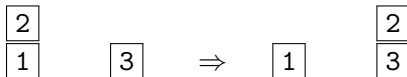
Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$  (piesa mutată e liberă)

$\neg p_{1on3} \wedge \neg p_{2on3}$  (piesa țintă e liberă)

Implicit:  $\neg p'_{2on0} \wedge \neg p'_{2on1}$  (2 nu va fi pe altceva)

## Reprezentarea unei tranziții



*Efectul* mutării:  $p'_{2on3}$  (notăm cu ' starea următoare)

Constrângeri de execuție (starea anterioară):

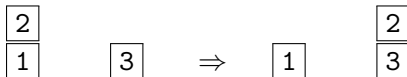
$\neg p_{1on2} \wedge \neg p_{3on2}$  (piesa mutată e liberă)

$\neg p_{1on3} \wedge \neg p_{2on3}$  (piesa țintă e liberă)

Implicit:  $\neg p'_{2on0} \wedge \neg p'_{2on1}$  (2 nu va fi pe altceva)

$\wedge \neg p'_{1on2} \wedge \neg p'_{3on2}$  (nu va fi altceva pe 2)

## Reprezentarea unei tranziții



*Efectul* mutării:  $p'_{2on3}$  (notăm cu ' starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$  (piesa mutată e liberă)

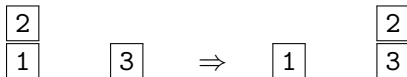
$\neg p_{1on3} \wedge \neg p_{2on3}$  (piesa țintă e liberă)

Implicit:  $\neg p'_{2on0} \wedge \neg p'_{2on1}$  (2 nu va fi pe altceva)

$\wedge \neg p'_{1on2} \wedge \neg p'_{3on2}$  (nu va fi altceva pe 2)

$\wedge \neg p'_{1on3}$  (nu va fi altceva pe 3)

## Reprezentarea unei tranziții



*Efectul* mutării:  $p'_{2on3}$  (notăm cu ' starea următoare)

Constrângeri de execuție (starea anterioară):

$\neg p_{1on2} \wedge \neg p_{3on2}$  (piesa mutată e liberă)

$\neg p_{1on3} \wedge \neg p_{2on3}$  (piesa țintă e liberă)

Implicit:  $\neg p'_{2on0} \wedge \neg p'_{2on1}$  (2 nu va fi pe altceva)

$\wedge \neg p'_{1on2} \wedge \neg p'_{3on2}$  (nu va fi altceva pe 2)

$\wedge \neg p'_{1on3}$  (nu va fi altceva pe 3)

Valorile rămân la fel pentru perechile neimplicate:

$p'_{1on0} = p_{1on0} \wedge p'_{3on0} = p_{3on0} \wedge p'_{3on1} = p_{3on1}$

Conjunția relațiilor descrie mutarea lui 2 pe 3, în toate cazurile

# Reprezentarea sistemului

*Spațiul* (mulțimea) *stărilor*  $S$ :

dat de propozițiile boolene  $p_{i \text{ on } j}$ ,  $1 \leq i \leq n, 0 \leq j \leq n, i \neq j$

*Relația de tranziție*:

se poate executa *oricare* (SAU) din tranzițiile posibile într-o stare:

$$\begin{aligned} & p'_{1 \text{ on } 0} \text{ (mută 1 pe masă)} \wedge \text{constrângeri mutare 1 pe 0} \\ \vee & p'_{1 \text{ on } 2} \text{ (mută 1 pe 2)} \wedge \text{constrângeri mutare 1 pe 2} \\ \vee & \dots \text{ (total } 3 \times 3 \text{ mutări potențiale)} \\ \vee & p'_{3 \text{ on } 2} \text{ (mută 3 pe 2)} \wedge \text{constrângeri mutare 3 pe 2} \end{aligned}$$

Notăm cu  $\bar{v} = \langle p_1, p_2, \dots, p_N \rangle$  vectorul de stare

Relația de tranziție e o formulă  $R(\bar{v}, \bar{v}')$

între starea curentă și starea următoare

! *Stările* și *tranzițiile* sistemului se reprezintă ca *formule logice*



## Găsirea unui plan

Fie  $S_0(\bar{v})$  și  $S_f(\bar{v})$  formulele ce exprimă stările inițiale și finale

Atingerea lui  $S_f$  din  $S_0$  în **1 mutare** = e realizabilă formula

$$S_0(\bar{v}_0) \wedge R(\bar{v}_0, \bar{v}_1) \wedge S_f(\bar{v}_1)$$

( $\bar{v}_0$  e o stare inițială și  $\bar{v}_1$  o stare finală și e o tranziție între ele)

Atingerea lui  $S_f$  din  $S_0$  în **k mutări** = e realizabilă formula

$$S_0(\bar{v}_0) \wedge R(\bar{v}_0, \bar{v}_1) \wedge \dots \wedge R(\bar{v}_{k-1}, \bar{v}_k) \wedge S_f(\bar{v}_k)$$

$\Rightarrow$  Găsim un plan de lungime minimă căutând succesiv soluții pentru formule tot mai complexe:  $2 \cdot N, \dots, (k+1) \cdot N$  propoziții

Există și alți algoritmi dedicați planificării.

Aici am redus problema la o exprimare **simplă**, fundamentală:

**rezolvarea unei formule boolene**

# Cum stabilim dacă o formulă e realizabilă ?

Observații și reguli simple:

1) Un literal *singur într-o clauză* are o singură valoare fezabilă:

în	$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$	$a$ trebuie să fie 1
în	$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c)$	$b$ trebuie să fie 0

# Cum stabilim dacă o formulă e realizabilă ?

Observații și reguli simple:

1) Un literal *singur într-o clauză* are o singură valoare fezabilă:

în  $a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$   $a$  trebuie să fie 1

în  $(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c)$   $b$  trebuie să fie 0

2 a) Dacă un literal e 1, *pot fi șterse clauzele* în care apare

b) Dacă un literal e 0, *el poate fi șters* din clauzele în care apare

Exemplele de mai sus se simplifică:

$$a \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \xrightarrow{a=1} (b \vee c) \wedge (\neg b \vee \neg c)$$

$$(a \vee b) \wedge \neg b \wedge (\neg a \vee \neg b \vee c) \xrightarrow{b=0} a \wedge (\neg a \vee c)$$

$a \wedge (\neg a \vee c)$  se simplifică mai departe la  $a = c = 1$

$\Rightarrow$  *formula e realizabilă*

## Cum stabilim dacă o formulă e realizabilă ?

3) Dacă *nu mai sunt clauze*, am terminat (și avem o atribuire)

Dacă se ajunge la o *clauză vidă*, formula *nu e realizabilă*

$$a \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

$$\xrightarrow{a=1} b \wedge (\neg b \vee c) \wedge (\neg b \vee \neg c)$$

$$\xrightarrow{b=1} c \wedge \neg c \quad \xrightarrow{c=1} \emptyset \quad (\neg c \text{ devine clauza vidă} \Rightarrow \text{nerealizabilă})$$

Dacă *nu mai putem face reduceri* după aceste reguli ?

$$a \wedge (\neg a \vee b \vee c) \wedge (\neg b \vee \neg c) \quad \xrightarrow{a=1} (b \vee c) \wedge (\neg b \vee \neg c) \quad ??$$

4) Alegem o variabilă și încercăm (*despărțim pe cazuri*)

- ▶ cu valoarea 1
- ▶ cu valoarea 0

O soluție pentru oricare caz e bună (nu căutăm o soluție anume).

Dacă nici un caz nu are soluție, formula nu e realizabilă.

# Un algoritm de rezolvare

Problema are ca date:

- ▶ lista clauzelor (formula)
- ▶ mulțimea variabilelor deja atribuite (inițial vidă)

Regulile 1 și 2 ne *reduc problema la una mai simplă*

(mai puține necunoscute sau clauze mai puține și/sau mai simple)

Regula 3 spune când ne oprim (avem răspunsul).

Regula 4 reduce problema la rezolvarea a *două probleme mai simple*  
(cu o necunoscută mai puțin)

Reducerea problemei la *aceeași problemă cu date mai simple*

(una sau mai multe instanțe) înseamnă că problema e *recursivă*.

Obligatoriu: trebuie să avem și o *condiție de oprire*

## Algoritmul Davis-Putnam-Logemann-Loveland

```
function solve(env: lit set, org-clauses: lit list list)
  clauses = simplify(org-clauses, env)
  if clauses is empty list then
    return true;
  if clauses has empty clause then
    return false;
  if clauses contains single literal a then
    solve (env with a=true, clauses)
  else
    a = choose-literal
    if solve (env with a=false, clauses) then
      return true;
    else if solve (env with a=true, clauses) then
      return true;
    else
      return false;
```

Cu optimizări poate rezolva formule cu  $10^4 - 10^5$  variabile

# Implementare: lucrul cu liste și mulțimi

Structuri de date:

- ▶ *lista* clauzelor (listă de liste de literale)
- ▶ *mulțimea* literalelor atribuite cu 1

Prelucrări:

# Implementare: lucrul cu liste și mulțimi

Structuri de date:

- ▶ *lista* clauzelor (listă de liste de literale)
- ▶ *mulțimea* literalelor atribuite cu 1

Prelucrări:

- ▶ *căutarea* unui literal în mulțimea celor atribuite



# Implementare: lucrul cu liste și mulțimi

Structuri de date:

- ▶ *lista* clauzelor (listă de liste de literale)
- ▶ *mulțimea* literalelor atribuite cu 1

Prelucrări:

- ▶ *căutarea* unui literal în mulțimea celor atribuite
- ▶ *adăugarea* unui literal la mulțimea celor atribuite

# Implementare: lucrul cu liste și mulțimi

Structuri de date:

- ▶ *lista* clauzelor (listă de liste de literale)
- ▶ *mulțimea* literalelor atribuite cu 1

Prelucrări:

- ▶ *căutarea* unui literal în mulțimea celor atribuite
- ▶ *adăugarea* unui literal la mulțimea celor atribuite
- ▶ *parcurgerea* literalelor dintr-o listă (clauză)

# Implementare: lucrul cu liste și mulțimi

Structuri de date:

- ▶ *lista* clauzelor (listă de liste de literale)
- ▶ *mulțimea* literalelor atribuite cu 1

Prelucrări:

- ▶ *căutarea* unui literal în mulțimea celor atribuite
- ▶ *adăugarea* unui literal la mulțimea celor atribuite
- ▶ *parcurgerea* literalelor dintr-o listă (clauză)
- ▶ *eliminarea* unui literal dintr-o listă (clauză)

# Implementare: lucrul cu liste și mulțimi

Structuri de date:

- ▶ *lista* clauzelor (listă de liste de literale)
- ▶ *mulțimea* literalelor atribuite cu 1

Prelucrări:

- ▶ *căutarea* unui literal în mulțimea celor atribuite
- ▶ *adăugarea* unui literal la mulțimea celor atribuite
- ▶ *parcurgerea* literalelor dintr-o listă (clauză)
- ▶ *eliminarea* unui literal dintr-o listă (clauză)
- ▶ *eliminarea* unei clauze dintr-o listă (formula)

⇒ avem nevoie de tipuri de date de nivel înalt și operații cu ele

# Limbajul OCaml (Objective Caml)

Un dialect obiectual al limbajului ML

Limbaj de programare *funcțional*

Cod compact, modular, parametrism polimorfic

Limbaj puternic tipizat  $\Rightarrow$  reduce mult erorile

Companii: Intel, Canon, Philips, Boeing, Sun

Microsoft: limbajul F# (.NET); verificatoare pentru programe

Wall Street: Jane St. Capital scrie programe *doar în OCaml*

Resurse: <http://caml.inria.fr>

IDE: OcaIDE <http://www.algo-prog.info/ocaide>

## Iterarea prin liste

env : mulțime de literale adevărate

lst : listă de literale (propoziții sau negații)

Dorim: crearea unei noi clauze din care ștergem literalele false

Simplificăm : selectăm elementele  $\geq 0$  dintr-o listă de întregi

```
let rec clrneg lst = match lst with
  [] -> []
  | h :: t -> if h >= 0 then h :: clrneg t else clrneg t
```

```
val clrneg : int list -> int list = <fun>
```

```
# clrneg [1; -4; 5; 6; -7; -4; 2];;
```

```
- : int list = [1; 5; 6; 2]
```

Important: s-a creat o *listă nouă*, fără a modifica cea veche  
(important în recursivitate: fiecare apel lucrează cu datele proprii)

## Iterarea prin liste

Funcția scrisă *filtrează* elemente după un anumit criteriu.

Pentru alt criteriu s-ar schimba doar testul ( $h \geq 0$ )

⇒ putem scrie prelucrarea parametrizând criteriul de filtrare:

```
let rec filter f lst = match lst with
  [] -> []
  | h :: t -> if f h then h :: filter f t else filter f t
(* sau, factorizand prelucrarea comuna *)
let rec filter f lst = match lst with
  [] -> []
  | h :: t -> let nt = filter f t in
    if f h then h :: nt else nt
```

Funcția *filter* există, predefinită în modulul *List*

⇒ nu e nevoie să rescriem prelucrarea recursivă a listei:

```
List.filter (fun x -> x >= 0) [1; -2; 3]
```

## Alte prelucrări iterative

**iter**: aplică o funcție la toate elementele listei

```
List.iter: ('a -> unit) -> 'a list -> unit = <fun>  
  List.iter (fun x -> print_int x) [1; -2; 3]
```

**map**: creează o listă nouă aplicând o funcție la elemente.

```
List.map: ('a -> 'b) -> 'a list -> 'b list = <fun>  
  List.map (fun x -> -x) [1; -2; 3]
```

Iteratorii sunt *polimorfici* (aplicabili la orice tip de liste).

Scriem direct funcțiile aplicate, nu e necesară definirea separată (funcțiile se folosesc ca și orice alte valori)

! Cu *iteratorii* scriem cod simplu, clar, corect, modular.



## E timpul să structurăm

Vrem cod independent de reprezentarea literalelor (șiruri, întregi...)

E esențial: să putem nega un literal, și să putem crea mulțimi.

Defini *semnătura* (interfața) unui tip și un modul de implementare

```
module type LITERAL = sig                                (* interfața *)
  type t
  val compare: t -> t -> int  (* necesar pt. mulțimi *)
  val negate: t -> t
end
module IntLit = struct  (* instantiem tipul propriu-zis *)
  type t = int          (* literal=identificator intreg *)
  let compare = Pervasives.compare  (* fct. standard *)
    (* sau: = fun x y -> x - y *)
  let negate x = -x
end
```

(cod după Conchon et. al, SAT-MICRO)

## Un modul parametrizat

Creem un modul care poate lucra cu orice literal care satisface interfața (semnătura) `LITERAL` definită.

⇒ putem schimba oricând reprezentarea, păstrând codul

```
module Sat(L: LITERAL) = struct

  module S = Set.Make(L)  (* tipul multime de literali *)

  (* aici definim functiile modulului *)

end
```

## Revenim: filtrăm literalele adevărate

Păstrăm în clauza `cl` doar `lit.` care *nu* apar negate în `env` (R2b)

```
List.filter (fun lit ->  
              not (S.mem (L.negate lit) env)) cl  
(* S.mem e functia membru pentru tipul multime S *)
```

Funcția transformă fiecare element din `clauses`  $\Rightarrow$  o nouă listă

```
List.map (fun cl -> List.filter  
          (fun lit ->  
            not (S.mem (L.negate lit) env)) cl) clauses
```

Găsirea unui literal adevărat e un caz special (R2a)

$\Rightarrow$  nu mai continuăm prelucrarea clauzei (*excepție*)

$\Rightarrow$  clauza e ștearsă  $\Rightarrow$  nu putem folosi `map`

## O mapare selectivă a listelor

Înlocuim map cu o funcție care poate elimina elemente din listă:

```
(* am denumit 'process' filtrarea definita anterior *)  
let rec filtermap clauses = match clauses with  
  [] -> []  
  | cl :: t -> let newcl = process cl in  
    if newcl = [] then filtermap t  
    else newcl :: filtermap t
```

Dacă newcl nu e vidă, e adăugată la lista rezultat.

Funcția face prelucrări (`::`) *după revenirea* din apelul recursiv

⇒ folosește stivă proporțională cu lungimea listei

Soluție: *acumularea* rezultatului ca parametru suplimentar

⇒ *recursivitate la dreapta* (tail recursion)

⇒ *transformabilă automat în ciclu*, nu consumă stiva

## De la map la o funcție mai generală: fold

```
(* am denumit 'process' prelucrarea unei liste-element *)  
let rec filtermap result clauses = match clauses with  
  [] -> result      (* rezultatul acumulat pana acum *)  
  | cl :: t -> let newcl = process cl in  
    if newcl = [] then filtermap result t  
    else filtermap (newcl :: result) t
```

Funcția se apelează recursiv pe coada listei *t* cu un *rezultat parțial* care e o funcție de cel anterior result și capul listei *cl*

```
let rec fold_left f res lst = match lst with  
  [] -> res  
  | h :: t -> fold_left f (f res h) t
```

$fold\_left\ f\ r\ [x_1; \dots; x_n] = f(\dots f(f(f\ r\ x_1)\ x_2)\ x_3 \dots)\ x_n$

Funcția e predefinită: `List.fold_left`

## Exemple cu fold\_left

*Suma* elementelor unei liste:

```
List.fold_left (fun x y -> x + y) 0 [1; 4; 6]
```

*Produsul* elementelor unei liste:

```
List.fold_left (fun x y -> x * y) 1 [2; 4; 5; 7]
```

*Inversarea* unei liste:

```
List.fold_left (fun t h -> h :: t) [] [1; 2; 3; 4]
```

Funcțiile *filter* și *map* sunt cazuri particulare:

```
let filter f lst = List.rev (List.fold_left  
    (fun r h -> if f h then h :: r else r) [] lst)  
let map f lst = List.rev  
    (List.fold_left (fun r h -> f h :: r) [] lst)
```

## Simplificarea clauzelor

Construim lista de clauze noi aplicând `fold_left` pe `clauses` pornind de la lista vidă:

```
let simplify (env, clauses) =  
  List.fold_left (  
    fun ncls cl -> (* ncls = lista clauze noi *)  
      let newcl = List.filter (* R2b *)  
        (fun lit -> not (S.mem (L.negate lit) env)) cl  
      in match newcl with  
        [] -> raise Unsat (* clauza vidă, R3 *)  
        | [lit] -> ncls (* șterge clauza unitate, R2a *)  
        | _ -> newcl :: ncls (* adaugă clauza modif. *)  
  ) [] clauses
```

Completăm codul:

dacă `filter` găsește `lit`. în `env`, ștergem clauza (excepție, R2a)  
un literal unitate `[lit]` e adăugat la `env` ca 1 (R1+R2a)

## Simplificarea clauzelor (cont.)

```
let simplify (env, clauses) =  
  List.fold_left (  
    fun (env, ncls) cl -> try  
      let newcl = List.filter  
        (fun lit ->  
          if S.mem lit env then raise Exit;  
          not (S.mem (L.negate lit) env)) cl  
      in match newcl with  
        [] -> raise Unsat  
        | [lit] -> assume lit (env, ncls)  
        | _ -> (env, newcl :: ncls)  
    with Exit -> (env, ncls)  
  ) (env, []) clauses
```

Noua variantă returnează o *pereche* (env, clauses)

assume creează un nou env cu lit=1 (R1)

și apelează simplify pe clauze (R2): funcții *mutual recursive*

⇒ apelurile reciproce aplică R1, R2 până nu mai apar modificări



## Simplificarea clauzelor (completată)

```
let rec assume lit (env, clauses) =  
  if S.mem lit env then (env, clauses)    (* neschimbat *)  
  else simplify ((S.add lit env), clauses)  (* R2(R1) *)  
and simplify (env, clauses) =  
  List.fold_left (  
    fun (env, ncls) cl -> try  
      let newcl = List.filter  
        (fun lit ->  
          if S.mem lit env then raise Exit;  
          not (S.mem (L.negate lit) env)) cl  
      in match newcl with  
        [] -> raise Unsat  
        | [lit] -> assume lit (env, ncls)  
        | _ -> (env, newcl :: ncls)  
    with Exit -> (env, ncls)  
  ) (env, []) clauses
```

## Verificarea propriu-zisă

Implementăm regula 4 care încearcă ambele valori pentru un literal:

```
let rec cksat (env, clauses) =  
  match clauses with  
  | [] -> true (* nu mai sunt clauze *)  
  | ([_] | []): _ -> assert false (* caz imposibil *)  
  | (a :: cl1) :: rest -> checksat (assume a (env, rest))  
    or cksat (assume (L.negate a) (env, cl1 :: rest))  
let sat clauses = (* încercare de la env vid *)  
  try cksat (simplify (S.empty, clauses)) with Unsat -> false
```

Clauză vidă sau unitară nu pot apare, sunt tratate de R2 (simplify)

Dar a rămas netratată excepția Unsat din assume ( $\rightarrow$  false)  
 $\Rightarrow$  înlocuim apelul la assume cu o funcție care tratează excepția

```
let wrapchoice lit (env, clauses) =  
  try cksat (assume lit (env, clauses)) with Unsat -> false
```

cksat și wrapchoice sunt mutual recursive  $\Rightarrow$  se declară împreună

## Soluția finală: funcția sat

```
let rec cksat (env, clauses) =  
  match clauses with  
  | [] -> true  
  | ([_] | [_]) :: _ -> assert false  
  | (a :: cl1) :: rest -> wrapchoice a (env, rest)  
    or wrapchoice (L.negate a) (env, cl1 :: rest)  
and wrapchoice lit (env, clauses) =  
  try cksat (assume lit (env, clauses)) with Unsat -> false  
  
let sat clauses =  
  (* rezultatul boolean *)  
  try cksat (simplify (S.empty, clauses)) with Unsat -> false  
  
module SI = Sat(IntLit) (* instanțiază modulul *)  
Printf.printf "%b" (SI.sat [[1;2;3];[-1;2;4];[2;3;-4]])
```

Exercițiu: scrieți o funcție alternativă care furnizează o atribuire (env) care satisface o formulă.